University of Zurich[UZH]

Department of Informatics

Interval-Dependent Attributes in Relational Database Systems

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology
of the University of Zurich

for the degree of
Doctor of Science (Ph.D.)

by

Anton Dignös

from Tramin, BZ, Italy

Accepted on the recommendation of

Prof. Dr. Michael H. Böhlen
Prof. Dr. Christian S. Jensen

2014

University of Zurich[UZH]

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, October 22, 2014

Head of the Ph.D. committee for informatics: Prof. Dr. Elaine M. Huang

Abstract

Data with time intervals is prominently present in finance, accounting, medicine and many other application domains. When querying such data, it is important to perform operations on aligned intervals, i.e., data is processed together only for the common interval where it is valid in the real world. For instance, an employee contributed to a project only for the time period where both the project was running and the employee was employed by the company, i.e., the employee contributed to the project only over their aligned time interval. A temporal join is thus only evaluated over the aligned interval of an employee and a project.

The problem of performing temporal operations, such as temporal aggregation or temporal joins, on data with time intervals using relational database systems can be attributed to the lack of primitives for the alignment of intervals. Even more challenges arise, when the data includes attribute values that are interval-dependent, such as project budgets or cumulative costs, and need to be scaled along with the alignment of intervals during processing. The goal of this thesis is to provide systematic and built-in support for querying data with intervals in relational database systems.

The solution we propose uses two temporal primitives a *temporal normalizer* and a *temporal aligner* for the alignment of intervals. Temporal operators on interval data are defined by *reduction rules* that map a temporal operator to an operation with a temporal primitive followed by the corresponding traditional non-temporal operator that uses equality on aligned intervals. A key feature of our approach is that operators can access the original time intervals in predicates

and functions, such as join conditions and aggregation functions, using *timestamp propagation*. Our approach, through timestamp propagation, supports the scaling of attribute values that are interval-dependent. When intervals are aligned during query processing, scaling can be performed at query time with the help of user-defined functions. This allows users to choose whether and how attribute values should be scaled. This is necessary since they may be interested in the total value in one query and the scaled value according to days or even working days in another query.

We integrated our solution into the kernel of the open source database system PostgreSQL, which allows to leverage existing query optimization techniques and algorithms.

Zusammenfassung

Daten mit Zeitintervallen spielen eine wesentliche Rolle in der Buchhaltung, Finanzwirtschaft und Medizin, sowie in vielen anderen Bereichen. Um solche Daten abzufragen, ist es wichtig Operationen über angeglichenen Zeitintervallen auszuführen, damit Daten nur über das Zeitintervall verarbeitet werden, in dem sie auch gültig sind. Zum Beispiel kann ein Angestellter an einem Projekt nur über jenen Zeitraum gearbeitet haben, für welchen beide, das Projekt und die Anstellung gültig waren, d.h. ein Angestellter trägt zu einem Projekt nur über das angeglichene gemeinsame Zeitintervall bei. Ein temporaler Join wird deshalb nur über das angeglichene Zeitintervall verarbeitet.

Das Problem bei der Verarbeitung temporaler Operationen in relationalen Datenbanken, wie z.B. temporale Aggregation oder temporaler Join, über Daten mit Zeitintervallen liegt an der unzureichenden Unterstützung von Primitiven für das Angleichen von Zeitintervallen. Eine daraus resultierende Herausforderung sind Daten mit Attributwerten, welche vom Zeitintervall abhängig sind, wie z.B. Projektbudgets oder kumulative Kosten, und aufgrund des Angleichens der Zeitintervalle, von den alten auf die neuen Zeitintervalle skaliert werden müssen. Die Zielsetzung dieser Dissertation ist es, eine systematische Unterstützung für Abfragen über Daten mit Zeitintervallen in den Kern von relationalen Datenbanken zu integrieren.

Unsere Lösung verwendet zwei temporale Primitiven, einen *temporal normalizer* und einen *temporal aligner*, um Zeitintervalle anzugleichen. Temporale Operatoren für Daten mit Zeitintervallen sind durch *reduction rules* definiert, welche einen temporalen Operator auf eine temporale

Primitive, gefolgt von einem herkömmlichen Datenbank Operator, reduzieren. Ein besonderes Merkmal unseres Ansatzes ist, dass Prädikate und Funktionen von temporalen Operatoren mit Hilfe von *timestamp propagation* auf die originalen Zeitintervalle von Tupeln zugreifen können, wie z.B. in Joinprädikaten und Aggregatsfunktionen.

Zusätzlich unterstützt unser Ansatz, mit Hilfe von timestamp propagation, das Skalieren von Attributwerten, welche vom Zeitintervall abhängig sind. Wenn ein Intervall bei der Verarbeitung angeglichen wird, kann die Skalierung mit Hilfe von benutzerdefinierten Funktionen zur Laufzeit einer Abfrage durchgeführt werden. Dies ermöglicht den Benutzern zu entscheiden, wann und wie Attributwerte skaliert werden sollen. Diese Eigenschaft ist notwendig, da Benutzer in manchen Abfragen den gesamten Wert und in anderen Abfragen den skalierten Wert anhand von Tagen oder Arbeitstagen haben möchten.

Wir haben unsere Lösung in das Open Source Datenbanksystem PostgreSQL integriert, was uns erlaubt, bereits existierende Anfrageoptimierungen und Algorithmen wiederzuverwenden.

Dedicated in loving memory to my parents

Acknowledgments

I would like to express my sincerest gratitude to my advisor Prof. Michael Böhlen for his support, valuable advices and constructive criticism. I am thankful for all the time he spent with me in discussions, meetings and iterations over papers and presentations, which substantially helped to increase the quality of this work. Thank you for giving me the opportunity to pursue a PhD in your research group.

I thank my BSc and MSc thesis advisor and co-author Prof. Johann Gamper for the time and effort he spent on iterations and polishing of the papers and presentations.

A special thank to Prof. Christian S. Jensen for agreeing to co-advise this thesis and to Prof. Renato Pajarola for chairing the thesis defense.

I would also like to thank all my colleagues from the Database Technology Group at the University of Zurich for their help and discussions.

Many thanks also to the database guys from the Free University of Bozen-Bolzano and University of Salzburg for the constructive feedback at our annual database retreats.

Anton Dignös
Zurich, August 2014

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Interval Timestamped Temporal Databases

Data with time intervals is ubiquitous in applications that keep track of historical data. Examples are financial applications that associate intervals with budgets, accounting applications that associate intervals with personnel information, and hotel reservation systems that associate intervals with reservations.

Recently, the support for data with intervals has gained a lot of interest from database vendors and standardization institutes. Examples are temporal table support in Teradata 13.10 [Ter10] (2010), period specifications in the SQL:2011 standard [KM12] (2011), which has been implemented in IBM DB2 10 [SNG12], and range types in PostgreSQL as of version 9.2 [Gro12] (2012). The major database system companies have added the infrastructure to store and manage intervals using predicates and functions. The support for processing this data in a principled way however is rather limited. This forces application programmers to develop the logic outside of the database system, yielding less maintainable, error prone and inefficient applications.

This thesis is about (i) the integration of a comprehensive relational algebra for interval data inside a database system using temporal primitives; (ii) an approach to deal with attribute values that are interval-dependent; and (iii) a partition join for interval data that is effective for cases when other join algorithms do not perform well.

Instead of developing individual algorithms for each operator of the relational algebra for interval data, our approach is to have a small number of temporal primitives that encapsulate the splitting of intervals required for processing, and then use traditional operators to compute the result. This allows to leverage a large part of the query optimizer such as join reordering or selection-pushdown, and a large number of algorithms, such as join or aggregation algorithms. The integration of this temporal primitives into the database kernel compared to middleware implementations or approaches based on user-defined table functions, has the advantage that they are directly integrated into the pipelining mechanism of the database system, i.e., results do not have to be written back to the database system for further querying, and the primitives are not a black-box to the query optimizer, such as table functions, but give for instance estimates on cost and number of tuples they return, which is then used for selecting evaluation algorithms.

The rest of this section gives a running example that is used to illustrate the contributions of this thesis.

Example 1. Assume two company databases each with a project relation $\mathbf{P_1}$ and $\mathbf{P_2}$, respectively, as shown in Figure 1.1. The project relations consists of a project name, the budget and the time period when the project runs. For instance, tuple r_1 records that a project with name X runs from February 2014 till June 2014 with a budget of 75K.

$\mathbf{P_1}$

	N_1	B_1	T
r_1	X	75K	[2014-02, 2014-07)
r_2	Y	40K	[2014-06, 2014-10)

$\mathbf{P_2}$

	N_2	B_2	T
s_1	A	70K	[2014-01, 2014-08)
s_2	B	30K	[2014-02, 2014-04)
s_3	C	40K	[2014-09, 2015-01)
s_4	D	20K	[2014-11, 2015-01)

Figure 1.1: Project Relations of Two Companies.

Assume one company wants to compare its project budgets with the project budgets of the other company that run concurrently, and have an overlap for at least 2 months. Project periods for which no such comparison is available should be returned as well. This analysis corresponds to a temporal left outer join of the two project relations, with the condition that the overlapping time

interval is at least 2 months long and the budget over the entire project period is scaled to the common time period. The algebra operation for this query is as follows:

$$\mathbf{Z} \leftarrow \mathbf{P_1} \overset{T:scale(B_1),scale(B_2)}{\underset{DUR(\mathbf{P_1}.T \cap \mathbf{P_2}.T) \geq 2\text{months}}{\rtimes\!\!\bowtie}} \mathbf{P_2}.$$

The ^{T} indicates that the left outer join ($\rtimes\!\!\bowtie$) is a temporal left outer join, where only overlapping (concurrent) project periods are considered, the condition $_{DUR(\mathbf{P_1}.T \cap \mathbf{P_2}.T) \geq 2\text{months}}$ is the additional restriction that the overlapping interval is at least two months, and $^{:scale(B_1),scale(B_2)}$ specifies that the budget values of the projects need to be scaled from the original project period to the overlapping time period. The result is shown in Figure 1.2. For instance, tuple z_1 compares the budget

Z

	N_1	B_1	N_2	B_2	T
z_1	X	75K	A	50K	[2014-02, 2014-07)
z_2	X	30K	B	30K	[2014-02, 2014-04)
z_3	Y	20K	A	20K	[2014-06, 2014-08)
z_4	Y	20K	ω	ω	[2014-08, 2014-10)

Figure 1.2: Comparison of Concurrent Budget of Projects that Run Concurrently for at Least Two Months.

of project X with the budget of project A over their concurrent time period [2014-02, 2014-07). The budget of project X for this time period is 75K, since the project's original time period is exactly this period. The budget of project A over this time period is 50K, i.e., its total budget of 70K scaled to the new (smaller) time period. As a scaling function for this example we use uniform distribution over months, i.e., 70K is scaled from a 7 months period to a 5 months period resulting in 50K. Tuple z_4 shows the budget of project Y over time period [2014-08, 2014-10) for which no matching project in the other relation exists that runs concurrently for at least 2 months. In this case the result table shows ω (NULL) values, indicating that no comparison is available.

The challenges for traditional database systems to compute temporal operations on interval-timestamped data, such as in Example 1, lay in the manipulation, i.e., splitting, of the intervals. For instance, the tuple r_2 for project Y contributes to the result tuple z_3, but only for its sub-interval [2014-06, 2014-08) for which it overlaps tuple s_1 for project A. The remaining sub-interval of r_2, i.e., [2014-08, 2014-10), for which no matching tuple in the other relation exists contributes to the result tuple z_4. Thus, the time interval of r_2 has been split into two intervals, [2014-06, 2014-08) and [2014-08, 2014-10), during processing.

The approaches proposed in this thesis are based on the following three observations:

Observation 1 Applications that store and manipulate data with time intervals require database operators that operate over tuples with aligned time intervals, respect lineage information, and have access to the original time intervals.

Observation 2 If intervals in the input relation are split according to the intervals of the result, then traditional (non-temporal) database operators can be used to perform temporal operations by adding an equality predicate on the intervals that have been split.

Observation 3 An interval of a result tuple is produced from (a) the intersection of **two** overlapping and matching intervals, or (b) the intersection of **all** overlapping and matching intervals, or (c) the maximal sub-interval of an interval for which **no other** overlapping and matching tuple exists.

From Observation 1 we derive the definitions of temporal operators, i.e., given a temporal operator, such as the temporal aggregation operator, we derive from Observation 1 how the result of the operator must be defined using its corresponding non-temporal operator over concurrent time intervals. Assume our running example relation $\mathbf{P_2}$ and the temporal aggregation operator. For instance, when we want to count the number of projects using temporal aggregation, we get a count of 1 over time interval $[$2014-01$, $2014-02$)$, since we have one project over $[$2014-01$, $2014-02$)$ and a traditional aggregation would give a count of 1. For time interval $[$2014-02$, $2014-04$)$ we have a count of 2, since there are two projects over this period, and so on. Respecting the changes given by the interval boundaries, i.e., lineage information, plays an important role for some applications. Consider now that we want the sum of available budgets in $\mathbf{P_2}$, i.e., temporal aggregation with sum over the scaled budget. For period $[$2014-02$, $2014-04$)$ the sum of available budgets is 50K, i.e., 20K from project A plus 30K from project B over period $[$2014-02$, $2014-04$)$. This is the period where there are no changes to the sum of the available budgets and it can be freely allocated or distributed within this period. The interval of this result tuple cannot be larger, since otherwise project B would either not have started yet or would already have finished, and its budget would not be available then. It can also not be shortened, i.e., 25K for $[$2014-02$, $2014-03$)$ and 25K for $[$2014-03$, $2014-04$)$ as this would restrict the period the budget can be freely allocated or distributed.

From Observation 2 we derived the idea of temporal primitives and reduction rules. Given a temporal operator, first a temporal primitive is used to split the intervals of the tuples of the input relations accordingly, and second the corresponding non-temporal operator with equality on the split

intervals is applied, to compute the final result. The reduction rules define a systematic mapping from a temporal operator to a specific temporal primitive and the corresponding non-temporal operator. For instance, in our running example, assume a temporal primitive that aligns relation $\mathbf{P_1}$ in a way, such that it contains a tuple $\tilde{r_1}$ for r_1 with time interval $[2014\text{-}02, 2014\text{-}07)$ and aligns relation $\mathbf{P_2}$ in a way, such that it contains a tuple $\tilde{s_1}$ for s_1 with time interval $[2014\text{-}02, 2014\text{-}07)$, then a traditional left outer join with equality on intervals, as it is available in any database system, produces the correct join result for $\tilde{r_1}$ and $\tilde{s_1}$, i.e., the result tuple z_1.

From Observation 3 we derive that exactly two temporal primitives are general enough to provide the mapping, i.e., the reduction rules. The **temporal aligner** splits tuples according to (a) and (c), and the **temporal normalizer** splits tuples according to (b) and (c). For instance, in our running example we have a temporal left outer join. The intervals of the result for this operator are produced by either intersections of two matching tuples (a), i.e., result tuples z_1, z_2, and z_3 or maximal sub-intervals for which no other matching tuple exists (c), i.e., result tuple z_4. Thus, the temporal primitive for the temporal left outer join is the temporal aligner. Other operations where the temporal aligner primitive is used are all kind of joins, such as inner and anti joins. The temporal normalizer primitive, on the other hand, is used for the other operators such as projection, aggregation, or set-operation.

When intervals are aligned during processing, some attribute values, such as available budgets or running costs, are no longer valid for the aligned intervals. For instance, if project Y has an available budget of 40K over the time period $[2014\text{-}06, 2014\text{-}10)$, it does not mean that it has an available budget of 40K over period $[2014\text{-}06, 2014\text{-}08)$ and an available budget of 40K over period $[2014\text{-}08, 2014\text{-}10)$, since this would wrongly indicate a total available budget of 80K over its entire time period $[2014\text{-}06, 2014\text{-}08)$.

To solve this problem this thesis proposes an approach to scale such attribute values from the original time period to the aligned time period during query processing. The proposed approach is general and has the following key features: (i) scaling is available at query time and not part of the relations schema, so users can decide **whether** they want to apply scaling or not; (ii) scaling is performed with the help of user-defined functions, so users can decide **how** attribute values should be scaled.

For instance, in our running example we are interested in comparing the available budget of projects. Thus, we must scale the budget value. In another query we may be interested in comparing the total budget of projects, i.e., not apply scaling. When scaling is applied, the choice how to scale may vary from query to query as well. For instance, scaling a project budget

of 75K uniformly from $[2014\text{-}02, 2014\text{-}07)$ to $[2014\text{-}02, 2014\text{-}04)$ using months results in 30K, when using days results in 29.5K, and when non-uniform distribution is used the results may vary even more. The choice of the scaling function to use, often depends on the semantics of the attribute or on the query and is application dependent.

1.2 Contributions

This thesis makes three main contributions in the field of interval timestamped databases:

- It introduces temporal primitives for the manipulation of intervals and reduction rules that can be used to map a temporal operation to an operation with primitives and the corresponding non-temporal operation.
- It introduces an approach to flexibly scale attribute values that are interval-dependent at query time, along with the manipulation of the intervals, using user-defined functions.
- It introduces an efficient partition join for interval data, that does not deteriorate when the data contains long intervals.

The research methodology that has been adopted for each part of this thesis starts with a problem given from real world followed by an analysis and precise definition of the problem. The solution to a problem and its properties are studied and elaborated analytically and then implemented. Large parts of this thesis have been implemented into the open source database system PostgreSQL and made available as open source[1]. The implementation is extensively evaluated and compared with state-of-the-art approaches to confirm the analytical results of the solution.

The rest of this section elaborates the contributions of this thesis in more detail with examples.

1.2.1 Temporal Primitives and a Comprehensive Relational Algebra for Interval Data

The first contribution of this thesis is a relational algebra for interval data, that satisfies all three properties of the sequenced semantics [BJ09]: (1) *snapshot reducibility* ensures that the result of

[1] http://www.ifi.uzh.ch/dbtg/research/align.html

a temporal operator at any time point p is equal to the result of the corresponding non-temporal operator applied to the input that is valid at p; (2) *extended snapshot reducibility* allows the use of predicates and functions on intervals, such as for instance in join conditions; and (3) *change preservation* ensures that the interval-boundaries of the input are preserved in the result.

To produce this temporal relational algebra, we identified two temporal primitives, a temporal normalizer $\mathcal{N}$ and a temporal aligner Φ, that allow to reduce temporal operations to the corresponding non-temporal operations that are already available in database systems. The temporal algebra is produced by so-called reduction rules, that define a mapping from a given temporal operator ψ^T to the corresponding non-temporal operator ψ after a temporal primitive has been applied to its inputs, i.e., $\psi^T = (\mathcal{N} \mid \Phi) \rightarrow \psi$.

Example 2. Consider our project relations from Example 1, and the query $\mathbf{P_1} \rtimes^T_{true} \mathbf{P_2}$ (without scaling), i.e., compare the total project budgets among all concurrent projects. The operation we want to perform is a temporal left outer join, and its reduction consists of the following two steps:

1. Align both relations:
   $\tilde{\mathbf{P_1}} \leftarrow \mathbf{P_1} \Phi_{true} \mathbf{P_2}$ (Align relation $\mathbf{P_1}$ according to relation $\mathbf{P_2}$)
   $\tilde{\mathbf{P_2}} \leftarrow \mathbf{P_2} \Phi_{true} \mathbf{P_1}$ (Align relation $\mathbf{P_2}$ according to relation $\mathbf{P_1}$)

2. Perform the corresponding non-temporal operation with equality on aligned intervals, and post processing:
   $\mathbf{Z} \leftarrow \alpha(\tilde{\mathbf{P_1}} \rtimes_{true \wedge \tilde{\mathbf{P_1}}.T = \tilde{\mathbf{P_2}}.T} \tilde{\mathbf{P_2}})$

Figure 1.3 shows the input relations $\mathbf{P_1}$ and $\mathbf{P_2}$ and the corresponding aligned relations (from step 1) $\tilde{\mathbf{P_1}}$ and $\tilde{\mathbf{P_2}}$, respectively.

The alignment primitive $\mathbf{P_1} \Phi_{true} \mathbf{P_2}$ splits each tuple in $\mathbf{P_1}$ into all intersection intervals with each matching tuple in $\mathbf{P_2}$, and into all maximal sub-intervals that do not overlap with a matching tuple in $\mathbf{P_2}$. For instance, tuple s_1 is split into $\tilde{s_{1a}}$, $\tilde{s_{1b}}$ and $\tilde{s_{1c}}$. Tuples $\tilde{s_{1b}}$ and $\tilde{s_{1c}}$ are created for the intersection intervals with the respectively matching tuples r_1 and r_2. Tuple $\tilde{s_{1a}}$ is created for the maximal sub-interval that does not overlap a matching tuple in relation $\mathbf{P_1}$.

After both relations have been aligned, tuples that match for the join have equal intervals. For instance, tuple $\tilde{r_{1a}}$ and $\tilde{s_{1b}}$ in the aligned relations now have equal intervals and the non-temporal left outer join with equality on intervals produces the result tuple z_1. The tuple $\tilde{r_{2b}}$ was produced

$\mathbf{P_1}$

	N_1	B_1	T
r_1	X	75K	[2014-02, 2014-07)
r_2	Y	40K	[2014-06, 2014-10)

$\mathbf{P_2}$

	N_2	B_2	T
s_1	A	70K	[2014-01, 2014-08)
s_2	B	30K	[2014-02, 2014-04)
s_3	C	40K	[2014-09, 2015-01)
s_4	D	20K	[2014-11, 2015-01)

(a) Input Relations.

$\tilde{\mathbf{P}}_\mathbf{1}$

	N_1	B_1	T
$r\tilde{}_{1a}$	X	75K	[2014-02, 2014-07)
$r\tilde{}_{1b}$	X	75K	[2014-02, 2014-04)
$r\tilde{}_{2a}$	Y	40K	[2014-06, 2014-08)
$r\tilde{}_{2b}$	Y	40K	[2014-08, 2014-09)
$r\tilde{}_{2c}$	Y	40K	[2014-09, 2014-10)

$\tilde{\mathbf{P}}_\mathbf{2}$

	N_2	B_2	T
$s\tilde{}_{1a}$	A	70K	[2014-01, 2014-02)
$s\tilde{}_{1b}$	A	70K	[2014-02, 2014-07)
$s\tilde{}_{1c}$	A	70K	[2014-06, 2014-08)
$s\tilde{}_{2a}$	B	30K	[2014-02, 2014-04)
$s\tilde{}_{3a}$	C	40K	[2014-09, 2014-10)
$s\tilde{}_{3b}$	C	40K	[2014-10, 2015-01)
$s\tilde{}_{4a}$	D	20K	[2014-11, 2015-01)

(b) Aligned Input Relations (Step 1).

$\mathbf{Z}$

	N_1	B_1	N_2	B_2	T
z_1	X	75K	A	70K	[2014-02, 2014-07)
z_2	X	75K	B	30K	[2014-02, 2014-04)
z_3	Y	40K	A	70K	[2014-06, 2014-08)
z_4	Y	40K	ω	ω	[2014-08, 2014-09)
z_5	Y	40K	C	40K	[2014-09, 2014-10)

(c) Result Relation (Step 2).

Figure 1.3: Input Relations $\mathbf{P_1}$ and $\mathbf{P_2}$, Aligned Relations $\tilde{\mathbf{P}}_\mathbf{1}$ and $\tilde{\mathbf{P}}_\mathbf{2}$, and Result Relation $\mathbf{Z}$ for the Temporal Operator $\mathbf{P_1} \,\tilde{\bowtie}^T_{true} \mathbf{P_2}$.

by a maximal sub-interval that does not have a match in the other relation and the non-temporal left outer join with equality on intervals produces the result tuple z_4.

A key feature of the temporal relational algebra covered in this thesis is that predicates and functions of operators can access the original intervals of tuples. Examples are the join predicate in Example 1, or a join predicate that only compares projects with the same project duration, or an aggregation function that computes the average duration of projects. To make the original intervals of tuples available to such predicates and functions, despite the splitting of the intervals, we propose timestamp propagation. First, the interval timestamps of tuples are propagated as additional non-temporal attributes that are not affected by the splitting, and second the refer-

ence to interval timestamps in predicates and functions are substituted with the reference to the propagated interval timestamps.

Timestamp propagation is the approach used for scaling to provide the scaling function with the original interval timestamp of a tuple, as illustrated below.

1.2.2 Integration of User-Defined Scaling Functions for Interval-Dependent Attribute Values

The second contribution of this thesis is to provide precise procedures to scale, at query time, attribute values that are interval dependent, with the help of scaling functions. A scaling function is a user-defined function (UDF) that returns the scaled attribute value from three input parameters: the attribute value to be scaled, the new interval timestamp to which to scale, and the original interval timestamp from which to scale.

The key behind scaling is that all values for the three parameters of the scaling function need to be available at the same time. To scale attribute values during the processing of a temporal operator, the general approach to reduce a temporal operation as described in the previous section is extended by two additional steps. A first step where timestamp propagation is applied, i.e., a copy of the original interval timestamp is preserved before it is aligned, so it is available later on for the scaling function, and a second step to perform the actual scaling with scaling functions supplied by the user.

Example 3. Consider our query with scaling from Example 1. The approach to compute this query consists of the following four steps:

1. Propagate interval-timestamps:
   $\mathbf{P_3} \leftarrow \epsilon_U(\mathbf{P_1})$
   $\mathbf{P_4} \leftarrow \epsilon_V(\mathbf{P_2})$

2. Align both relations:
   $\tilde{\mathbf{P_1}} \leftarrow \mathbf{P_3} \Phi_{DUR(\mathbf{P_3}.T \cap \mathbf{P_4}.T) \geq 2\text{months}} \mathbf{P_4}$
   $\tilde{\mathbf{P_2}} \leftarrow \mathbf{P_4} \Phi_{DUR(\mathbf{P_3}.T \cap \mathbf{P_4}.T) \geq 2\text{months}} \mathbf{P_3}$

3. Perform the corresponding non-temporal operation with equality on aligned intervals, and post processing:
   $\mathbf{Z} \leftarrow \alpha(\tilde{\mathbf{P_1}} \rtimes_{DUR(U \cap V) \geq 2\text{months} \wedge \tilde{\mathbf{P_1}}.T = \tilde{\mathbf{P_2}}.T} \tilde{\mathbf{P_2}})$

4. Scale and remove propagated interval-timestamps:
   $\mathbf{Z} \leftarrow \pi_{N_1,scale(B_1,T,U)/B_1,N_2,scale(B_2,T,V)/B_2,T}(\mathbf{Z})$

Figure 1.4 shows the result of step 1, i.e., timestamp propagation. Each input relation $\mathbf{P_1}$ and $\mathbf{P_2}$ has an additional attribute U and V, respectively, that is a copy of the interval timestamp. These propagated timestamps are used later in the join condition, which accesses the interval timestamps, and for scaling. Figure 1.5 shows the result after the input relations have been

$\mathbf{P_3}$

	N_1	B_1	T	U
r_1	X	75K	[2014-02, 2014-07)	[2014-02, 2014-07)
r_2	Y	40K	[2014-06, 2014-10)	[2014-06, 2014-10)

$\mathbf{P_4}$

	N_2	B_2	T	V
s_1	A	70K	[2014-01, 2014-08)	[2014-01, 2014-08)
s_2	B	30K	[2014-02, 2014-04)	[2014-02, 2014-04)
s_3	C	40K	[2014-09, 2015-01)	[2014-09, 2015-01)
s_4	D	20K	[2014-11, 2015-01)	[2014-11, 2015-01)

Figure 1.4: Input Relations $\mathbf{P_1}$ and $\mathbf{P_2}$ with Propagated Interval Timestamps (Step 1).

aligned (step 2). Note that the copies of the original interval timestamps U and V are still available, and have not been aligned. The result of step 3, i.e., the non-temporal left outer join

$\tilde{\mathbf{P}}_\mathbf{1}$

	N_1	B_1	T	U
$\tilde{r_{1a}}$	X	75K	[2014-02, 2014-07)	[2014-02, 2014-07)
$\tilde{r_{1b}}$	X	75K	[2014-02, 2014-04)	[2014-02, 2014-07)
$\tilde{r_{2a}}$	Y	40K	[2014-06, 2014-08)	[2014-06, 2014-10)
$\tilde{r_{2b}}$	Y	40K	[2014-08, 2014-10)	[2014-06, 2014-10)

$\tilde{\mathbf{P}}_\mathbf{2}$

	N_2	B_2	T	V
$\tilde{s_{1a}}$	A	70K	[2014-01, 2014-02)	[2014-01, 2014-08)
$\tilde{s_{1b}}$	A	70K	[2014-02, 2014-07)	[2014-01, 2014-08)
$\tilde{s_{1c}}$	A	70K	[2014-06, 2014-08)	[2014-01, 2014-08)
$\tilde{s_{2a}}$	B	30K	[2014-02, 2014-04)	[2014-02, 2014-04)
$\tilde{s_{3a}}$	C	40K	[2014-09, 2015-01)	[2014-09, 2015-01)
$\tilde{s_{4a}}$	D	20K	[2014-11, 2015-01)	[2014-11, 2015-01)

Figure 1.5: Aligned Relations with Propagated Interval Timestamps (Step 2).

with equality on aligned intervals, and post processing is shown in Figure 1.6a. In this step the

references to the interval timestamps in the join condition are substituted with the references to propagated interval timestamps that have not been aligned, i.e., $\mathbf{P_1}.T$ is substituted with U and $\mathbf{P_2}.T$ is substituted with V. Note that after this step the entire information required for scaling is available, i.e., the original value for the budget attribute that needs to be scaled, the new interval timestamp to which we need to scale, and the original interval timestamp from which we need to scale. Finally, in step 4 the attribute values of the budget attribute are scaled to the new interval timestamps where for this example we used a scaling function that scales the budget uniformly according to months, and the original interval timestamps U and V are removed, since they are not needed any more. The final result is shown in Figure 1.6b, which is exactly the result of Example 1.

Z

	N_1	B_1	N_2	B_2	T	U	V
z_1	X	75K	A	70K	[2014-02, 2014-07)	[2014-02, 2014-07)	[2014-01, 2015-08)
z_2	X	75K	B	30K	[2014-02, 2014-04)	[2014-02, 2014-07)	[2014-02, 2014-07)
z_3	Y	40K	A	70K	[2014-06, 2014-08)	[2014-06, 2014-10)	[2014-01, 2014-08)
z_4	Y	40K	ω	ω	[2014-08, 2014-10)	[2014-06, 2014-10)	ω

(a) Result Relation before Scaling (Step 3).

Z

	N_1	B_1	N_2	B_2	T
z_1	X	75K	A	50K	[2014-02, 2014-07)
z_2	X	30K	B	30K	[2014-02, 2014-04)
z_3	Y	20K	A	20K	[2014-06, 2014-08)
z_4	Y	20K	ω	ω	[2014-08, 2014-10)

(b) Result Relation after Scaling (Step 4).

Figure 1.6: Result Relation before and after Scaling.

1.2.3 Efficient Partition Join for Interval Data

The third contribution of this thesis is an efficient partition join algorithm that computes the overlap join between two interval timestamped relations. The overlap join between two interval timestamped relations returns all pairs of tuples that have overlapping intervals. The efficient evaluation of this join is important to give the query optimizer an option when other predicates in the join are absent, exhibit a poor selectivity due to long histories, or must be evaluated after the overlapping interval has been computed, such as for instance the join predicate in our run-

ning example (Example 1). An efficient partitioning of interval-timestamped data is not only important for the temporal join as a stand-alone operation, but also for our temporal alignment primitive, which is used for the reduction of all kind of temporal joins, and relies on an efficient retrieval of pairs of tuples with overlapping intervals.

The proposed overlap interval partition join (OIPJOIN) is a partition join based on a novel partitioning approach termed overlap interval partitioning ($\mathcal{OIP}$). $\mathcal{OIP}$ divides the time range of a relation into k granules of equal duration. Partitions in $\mathcal{OIP}$ can have overlapping partition intervals that cover any sequence of adjacent granules. A tuple is placed in the partition for which the partition interval best fits the interval of the tuple. As a result intervals of tuples are partitioned by position and duration and tuples that behave similar during the join, i.e., tuples that join the same tuples of the other relation, are in the same partition.

The chosen number of granules k for $\mathcal{OIP}$ on one hand directly influences the number of false hits, i.e., the number of tuples that do not match during the join, and on the other hand influences the number of relevant partitions that need to be accessed during the join. Both factors incur CPU and IO cost and are inversely related. Increasing the number of granules k, decreases the number of false hits and thus decreases the CPU cost for comparisons and the IO cost for retrieving less data, but increases the number of partition accesses and thus the CPU cost for navigation in the access structure and the IO cost for partially filled storage block accesses.

To determine the optimal number of granules k for the OIPJOIN we analytically determined the average false hit ratio (AFR) and the average number of partition accesses (APA) depending on k. We construct the cost function using these measures and the CPU and IO cost, and minimize the cost function w.r.t. k. As a result the OIPJOIN is self-adjusting, i.e., given the size of the two relations to join and the cost for CPU and IO operations, it automatically determines the optimal number of granules k to partition the two relations. This makes the OIPJOIN adequate for the integration into a general purpose database system that cannot rely on user-specified or application specific parameters for the choice of the partitioning.

1.3 Organization of the Thesis

This thesis is based on a collection of papers. A bibliography for all chapters is given at the end of the thesis.

Chapter 2 Temporal Alignment

Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal Alignment. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 433–444, ACM, 2012.
DOI: http://dx.doi.org/10.1145/2213836.2213886

Chapter 3 Query Time Scaling of Attribute Values

Anton Dignös, Michael H. Böhlen, and Johann Gamper. Query Time Scaling of Attribute Values in Interval Timestamped Databases. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (Demonstration)*, ICDE '13, pages 1304–1307, IEEE Computer Society, 2013.
DOI: http://dx.doi.org/10.1109/ICDE.2013.6544930

Chapter 4 Overlap Interval Partition Join

Anton Dignös, Michael H. Böhlen, and Johann Gamper. Overlap Interval Partition Join. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1459–1470, ACM, 2014.
DOI: http://dx.doi.org/10.1145/2588555.2612175

CHAPTER 2

Temporal Alignment

Abstract

In order to process interval timestamped data, the sequenced semantics has been proposed. This paper presents a relational algebra solution that provides native support for the three properties of the sequenced semantics: snapshot reducibility, extended snapshot reducibility, and change preservation. We introduce two temporal primitives, *temporal splitter* and *temporal aligner*, and define rules that use these primitives to reduce the operators of a temporal algebra to their nontemporal counterparts. Our solution supports the three properties of the sequenced semantics through *interval adjustment* and *timestamp propagation*. We have implemented the temporal primitives and reduction rules in the kernel of PostgreSQL to get native database support for processing interval timestamped data. The support is comprehensive and includes outer joins, antijoins, and aggregations with predicates and functions over the time intervals of argument relations. The implementation and empirical evaluation confirms effectiveness and scalability of our solution that leverages existing database query optimization techniques.

2.1 Introduction

In order to query interval timestamped databases, temporal upward compatible, nonsequenced, and sequenced semantics exist [BJS00; BBJS97]. Temporal upward compatible semantics [BBJS97; BJS09b] processes only the data that is valid at the current time, whereas nonsequenced semantics [BJS09a] treats time intervals as conventional attributes. For both semantics standard SQL can be used to query the database. Sequenced semantics [BJ09] is by far the most difficult to support. Various works have shown that the formulation of sequenced statements in standard SQL is complex and awkward [BJS00; LSD$^+$01; Lor09; Sno00]. This paper proposes relational algebra primitives that provide support for the sequenced semantics, including outer joins, anti-joins and aggregations with predicates and functions over the interval timestamps of argument relations.

Sequenced semantics comes with three properties: snapshot reducibility, which applies nontemporal statements to each snapshot of a temporal database; extended snapshot reducibility, which combines snapshot reducibility with the possibility to specify predicates and functions over the interval timestamps of argument relations; and change preservation, which preserves the changes defined by the start and end points of time intervals.

Example 4. Consider a hotel that has rooms to let. Room prices are fixed during winter and negotiated during summer. The hotel has three fixed-price categories: short term (1-2 months, high price), long term (3-7 months, lower price) and permanent (8-12 months, lowest price and no summer/winter fluctuation). Room prices are recorded in relation $\mathbf{P}$, where A is the daily price, Min and Max are minimum and maximum duration for the specific price category, and T is the time period during which the price is valid. For instance, tuple s_1 records that short term guests pay a price of 50 during the first 5 months of 2012. During the same period long terms guests pay a price of 40 (tuple s_2). Tuple s_3 is for permanent guests with a price of 30 that is valid for the entire year. Relation $\mathbf{R}$ records reservations, where N is the name of the guest and T is the time period of a reservation. For instance, r_1 records a reservation of Ann for the first 7 months of 2012. r_3 records a different reservation for Ann from August until November 2012.

In order to determine periods with fixed prices and periods that need to be negotiated, the hotel computes a temporal left outer join: $Q1 = \mathbf{R} \,\tilde{⟕}^T_{Min \leq DUR(\mathbf{R}.T) \leq Max}\, \mathbf{P}$. The result of query $Q1$ is shown in Fig. 2.1b. We use a graphical representation, where timestamps are drawn as horizontal lines. For instance, $(\text{Ann}, 40, 3, 7, [2012/1, 2012/6))$ is produced from r_1 and s_1 over their common interval $[2012/1, 2012/6)$, and $(\text{Ann}, \omega, \omega, \omega, [2012/6, 2012/8))$ from r_1 over the

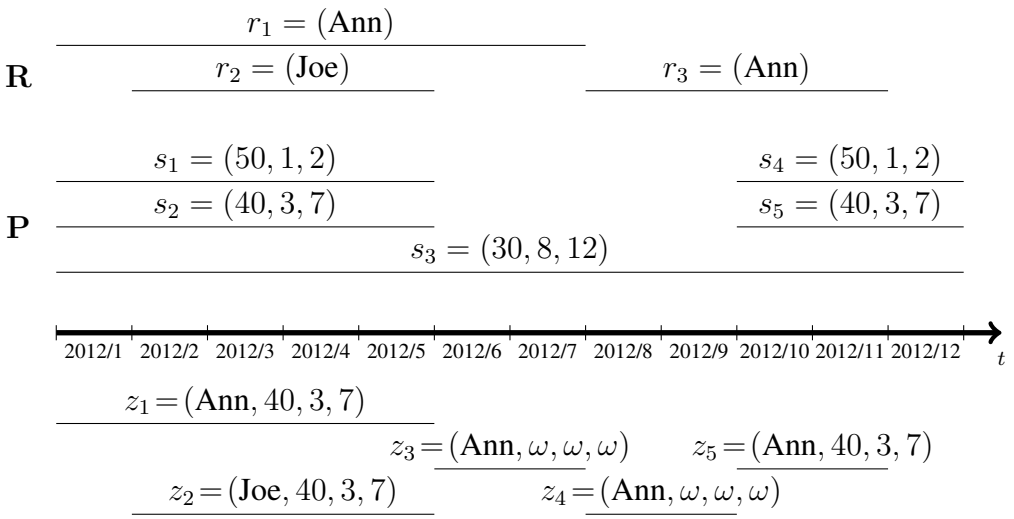
R

	N	T
r_1	Ann	$[2012/01, 2012/08)$
r_2	Joe	$[2012/02, 2012/06)$
r_3	Ann	$[2012/08, 2012/12)$

P

	A	Min	Max	T
s_1	50	1	2	$[2012/01, 2012/06)$
s_2	40	3	7	$[2012/01, 2012/06)$
s_3	30	8	12	$[2012/01, 2013/01)$
s_4	50	1	2	$[2012/10, 2013/01)$
s_5	40	3	7	$[2012/10, 2013/01)$

(a) Temporal Relations



(b) Result of Query $Q1$.

Figure 2.1: Sample Database.

interval $[2012/6, 2012/8)$ for which the price must be negotiated (ω denotes a null value). Note that the join predicate references the timestamp attribute $\mathbf{R}.T$ (extended snapshot reducibility) and that z_3 and z_4 are not coalesced into a single result tuple since they are derived from different argument tuples (change preservation).

In order to satisfy the three properties of the sequenced semantics, we propose a solution that (1) adjusts timestamps by breaking them into pieces, (2) propagates timestamps as explicit attributes to support functions and predicates over these intervals, and (3) uses lineage information to preserve the changes defined by the interval timestamps of the argument tuples. It is easy to support each property individually. Supporting all three properties together, however, is difficult and is the goal pursued in this paper.

To adjust interval timestamps, we propose two primitives that transform each tuple of an argument relation into a set of tuples with adjusted timestamps. Based on the characteristics of

how relational operators produce result tuples, we identify two classes of operators that have to be adjusted differently. For *group based* operators, $\{\pi, \vartheta, \cup, -, \cap\}$, all tuples in a group contribute to one result tuple. We define a *temporal splitter* to adjust interval timestamps for group based operators. For *tuple based* operators, $\{\sigma, \times, \bowtie, ⟕, ⟗, ⟖, \triangleright\}$, at most one tuple of every argument relation contributes to a single result tuple. We define a *temporal aligner* to adjust intervals for tuple based operators. Once the argument tuples have been adjusted, the final result can be computed by comparing interval timestamps using equality and without further interval manipulations.

The purpose of the adjustment is to modify interval timestamps, so that all intervals that have to be compared are either identical or disjoint. After the adjustment the original interval timestamps are no longer available. To permit the use of predicates over the original interval timestamps, we provide the possibility to *propagate timestamps* as explicit attributes. Timestamp propagation is possible for *schema robust* operators, i.e., operators that are not affected if the schema of an argument relation is extended with additional attributes (cf. Section 2.3.3). Apart from the set operators, $\{\cup, -, \cap\}$, all relational algebra operators are schema robust. In relational algebra expressions interval timestamps can be propagated through sequences of schema robust operators and used in predicates and functions. They must be removed (using a projection) before operators that are not schema robust.

Interval adjustment, in combination with timestamp propagation, provides a uniform solution to reduce all operators of a temporal algebra with sequenced semantics to their nontemporal counterparts. With the adjustment primitives query processing becomes a two-step process: 1) propagate and adjust the interval timestamps of argument tuples; 2) apply the corresponding nontemporal operator on the interval-adjusted relations.

To summarize, we adopt an interval based temporal data model and propose an algebraic solution that provides support for the sequenced semantics with snapshot reducibility, extended snapshot reducibility, and change preservation. Our solution makes it unnecessary for applications to explicitly manipulate intervals: tuples that have to be compared by relational algebra operators have either equal or disjoint timestamps. The technical contributions are as follows:

- We introduce *timestamp propagation* as a mechanism to support extended snapshot reducibility for *schema robust* operators, i.e., operators that are not affected if the schema is extended.

- We define *lineage* for interval timestamped databases and show how to combine lineage with snapshot reducibility to define *change preservation*.

- We define a *temporal splitter* and a *temporal aligner* primitive to adjust the timestamp intervals of argument tuples. The temporal splitter adjusts the intervals for the group based operators, $\{\pi, \vartheta, \cup, -, \cap\}$, the temporal aligner for the tuple based operators, $\{\sigma, \times, \bowtie, \rtimes, \ltimes, \rtimes, \triangleright\}$.

- We define a temporal algebra with sequenced semantics by specifying a set of *reduction rules* that reduce temporal operators to the nontemporal counterparts. The reduction rules are comprehensive and cover all algebra operators, including outer joins, antijoins, and aggregations with predicates and functions over the timestamp attributes.

- We prove that the temporal algebra defined by the reduction rules is snapshot reducible, extended snapshot reducible, and change preserving.

- We describe an implementation of the temporal primitives and reduction rules in the kernel of PostgreSQL and conduct extensive experiments that show the effectiveness and efficiency of our approach.

The rest of the paper is organized as follows. Section 2.2 discusses related work. Section 2.3 introduces the three properties of the sequenced semantics, including mechanisms and formalization to achieve them. In Sec. 2.4 we introduce temporal splitter and aligner, which in Sec. 2.5 are used to reduce the operators of a temporal algebra to their nontemporal counterparts. Section 2.6 describes the implementation of our solution in PostgreSQL. Section 2.7 reports the evaluation results. Section 2.8 concludes the paper and points to future work.

2.2 Related Work

The management of temporal data in DBMSs has been an active research area since several decades, focusing primarily on temporal data models and query languages (e.g., [AHVdB96; BJ02; DD02; JSS94; Sno95; BGJS09]) as well as efficient evaluation algorithms for specific operators (e.g., temporal join [Seg93; SSJ94] and temporal aggregation [BGJ06a; VLSM05; GBJ09]).

To make the formulation of temporal queries more convenient, various temporal query languages [BJS95; BJS00; SJS95] have been proposed. The earliest approach to add support for the time to relational query languages, such as SQL, was to introduce new data types with associated predicates and functions that were strongly influenced by Allen's interval relationships. Extending an existing query language with new data types is fairly simple and facilitates the formulation of some temporal queries. However, it does not provide a systematic way to generalize nontemporal to temporal queries since it does not effectively support, e.g., temporal aggregation and temporal set difference. In response to this new keywords and clauses were added to SQL with the goal of expressing temporal queries similar to nontemporal ones. Below we discuss the languages and techniques that are directly relevant to our solution. Note that the goal of our approach is not an extension of SQL, but native database support for temporal operators at the level of the relational algebra. Our solution is generic and provides built-in database support for implementing the proposed extensions of SQL.

The IXSQL language [DD02; LM97] normalizes timestamps and provides two functions, *unfold* and *fold*, that are used as follows: (i) unfold transforms an interval timestamped relation into a point timestamped relation by splitting each interval timestamped tuple into a set of point timestamped tuples; (ii) the corresponding nontemporal operator is applied on the normalized relation; (iii) fold collapses value-equivalent tuples over consecutive time points into interval timestamped tuples over maximal time intervals. The approach is conceptually simple, but timestamp normalization using fold and unfold does not preserve changes and an efficient implementation has not been provided.

An approach based on point timestamped relations is SQL/TP [Tom96; Tom97]. A temporal relation is a sequence of nontemporal relations (or snapshots), and the corresponding nontemporal operations are applied on each of the snapshots to answer temporal queries. To provide an efficient evaluation of SQL/TP an interval based encoding of point timestamped relations was proposed together with a *normalization* function. The normalization splits overlapping value-equivalent argument tuples into tuples with equal or disjoint timestamps and SQL/TP queries are then mapped to standard SQL statements with equality predicates. Toman's normalization function satisfies the properties of a temporal splitter for group based operators and we leverage the normalization for the splitting of interval timestamps of group based operators. We propose a database internal algorithm for the normalization function for which no implementation was provided. SQL/TP does not consider change preservation. Also not considered is extended snapshot reducibility, which is not relevant for point timestamped relations. Normalization is not applica-

ble to tuple based operators, such as joins, outer joins, and antijoins, since for these operators it would not preserve changes.

Agesen et al. [ABPT01] introduce a *split* operator that extends the normalization to bitemporal relations. The operator splits argument tuples that are value-equivalent over nontemporal attributes into tuples over smaller, yet maximal timestamps such that the new timestamps are either equal or disjoint. The nontemporal operations are applied to the split relation. Similar to our temporal splitter, changes are preserved. The focus of the split operator are aggregation and difference in now-relative bitemporal databases. It is limited to value-equivalent tuples, i.e., tuples with pairwise identical nontemporal attributes, and does not apply to change preserving joins, outer joins, and antijoins.

ATSQL [BJS00] offers a systematic way to construct temporal SQL queries from nontemporal SQL queries. The main idea is to formulate the nontemporal query and use *statement modifiers* to control if the statement is evaluated with temporal or nontemporal semantics. In the context of ATSQL different desiderata for temporal languages were formulated, namely upward compatible, temporal upward compatible, sequenced, and nonsequenced semantics. No native database implementation of this approach has been provided.

In terms of query processing various query evaluation algorithms for specific operators have been proposed. Join algorithms are based on indexing techniques [SE96; ZTS02] or well-known nested loop, sort merge and partitioning strategies [GJSS05]. Similarly, several solutions for the evaluation of various forms of temporal aggregation [BGJ06a; KS95; MFVLI03; YW03; ZMT+01] were proposed. Instead of designing algorithms for specific operators, we adopt a generic approach and propose two primitives that allow to reduce all operators of a sequenced algebra to their nontemporal counterparts.

The support for temporal data in commercial DBMSs has been limited to new data types with associated predicates and functions. In *PostgreSQL*, a temporal module [Dav11] adds the PERIOD datatype for anchored time intervals together with boolean predicates and functions, such as intersection, union and minus. Since not all of them are closed, the functions might throw a runtime error. While this module facilitates the formulation of some temporal queries, it does not conveniently support queries that need to adjust the timestamps of tuples, such as temporal difference, aggregation and outer joins. The *Oracle* database system [Mur08] extends the capabilities of PostgreSQL by additionally supporting valid and transaction time (DBMS_WM package). Querying temporal relations, however, is only possible at a specific time point (snapshot). *Teradata* [Ter10] provides similar temporal support as Oracle, i.e., the PERIOD datatype

with associated predicates and functions as well as valid time and transaction time. As of release 13.10, Teradata supports the temporal statement modifiers `SEQUENCED` and `NONSEQUENCED` in queries. The support for `SEQUENCED` is limited to inner joins. The sequenced semantics for outer joins, set operations, duplicate elimination and aggregation is not supported.

2.3 Sequenced Semantics

This section presents the three properties of the sequenced semantics [BJ09]: snapshot reducibility, extended snapshot reducibility and change preservation. For each property we provide crisp definitions that will be used to prove that our solution supports the sequenced semantics.

2.3.1 Preliminaries

We assume a linearly ordered, discrete time domain, Ω^T. A time interval is a contiguous set of time points (or instants) and is represented as a pair $[T_S, T_E)$, where T_S is the inclusive start point and T_E the exclusive end point. We use tuple timestamping and associate each tuple with a single time interval that represents the tuple's valid time. A temporal relation schema is represented as $R = (A_1, \ldots, A_m, T)$, where $A_1, \ldots, A_m$ are the nontemporal attributes with domain Ω_i and T is a temporal attribute over $\Omega^T \times \Omega^T$. Similarly, we assume a temporal relation $\mathbf{s}$ with schema $S = (C_1, \ldots, C_k, T)$. A tuple, r, over schema R is a finite set that contains for every A_i a value $v_i \in \Omega_i$ and for T a time interval $[T_S, T_E) \in \Omega^T \times \Omega^T$. A temporal relation, $\mathbf{r}$, over schema R is a finite set of tuples over R. For a tuple r and an attribute A_i, $r.A_i$ denotes the value of the attribute A_i in r. As abbreviations we use $\mathbf{A} = \{A_1, \ldots, A_m\}$ and $r.\mathbf{A} = (r.A_1, \ldots, r.A_m)$. The operators of the temporal relational algebra are selection σ^T, projection π^T, aggregation ϑ^T, difference $-^T$, union $\cup^T$, intersection $\cap^T$, Cartesian product $\times^T$, join $\bowtie^T$, left outer join $⟕^T$, right outer join $⟖^T$, full outer join $⟗^T$, and antijoin $\triangleright^T$. For the set operators we assume union compatible argument relations, and for $\pi^T_{\mathbf{B}}(\mathbf{r})$ and ${}_{\mathbf{B}}\vartheta^T_F(\mathbf{r})$ we require $\mathbf{B} \subseteq \mathbf{A}$. $sch(\psi)$ denotes the schema of the relation defined by the relational algebra expression ψ. We assume set-based semantics with duplicate free temporal relations, i.e., there are no value-equivalent tuples over common timepoints. Formally, a temporal relation, $\mathbf{r}$, is *duplicate free* iff $\forall r \in \mathbf{r} \forall r' \in \mathbf{r}(r \neq r' \Rightarrow r.\mathbf{A} \neq r'.\mathbf{A} \vee r.T \cap r'.T = \emptyset)$. A *snapshot* of a temporal relation is a nontemporal relation that is valid at a specific time point t, and is defined in terms of the timeslice operator [JS09]: $\tau_t(\mathbf{r}) = \{r.\mathbf{A} \mid r \in \mathbf{r} \wedge t \in r.T\}$.

2.3.2 Snapshot Reducibility

Many temporal languages [LM97; SJS95] are based on the concept of snapshot reducibility. Snapshot reducibility ensures that each snapshot in the result of a temporal operator (e.g., $\ltimes^T_\theta$) is equal to the result of the equivalent nontemporal operator (e.g., $\ltimes_\theta$) evaluated on the corresponding snapshots of the argument relations.

Definition 1. (*Snapshot Reducibility*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be temporal relations, ψ^T an n-ary temporal operator, ψ the corresponding nontemporal operator, Ω^T the time domain and $\tau_p(\mathbf{r})$ the timeslice operator. Operator ψ^T is *snapshot reducible* to ψ iff

$$\forall t \in \Omega^T(\tau_t(\psi^T(\mathbf{r}_1, \ldots, \mathbf{r}_n)) \equiv \psi(\tau_t(\mathbf{r}_1), \ldots, \tau_t(\mathbf{r}_n))).$$

Snapshot reducibility constrains the result of a temporal operator. Note that it does not define how to group time points into intervals, and the timestamps of the argument relations $\mathbf{r}_1, ..., \mathbf{r}_n$ cannot be used in theta conditions of ψ since the timestamps are removed by the timeslice operator (for instance, snapshot reducibility does not apply to $\vartheta^T_{AVG(DUR(R.T))}(\mathbf{R})$, which determines the average duration of reservations at each point in time).

2.3.3 Extended Snapshot Reducibility

As illustrated above, snapshot reducibility does not apply to temporal operators with predicates and functions over the interval timestamps of argument relations. The sequenced semantics introduces the concept of extended snapshot reducibility, which requires that references to interval timestamps can be used along with snapshot reducibility. We support extended snapshot reducibility by propagating interval timestamps as nontemporal attributes. Since timestamp propagation adds attributes to the schema of argument relations of an operator ψ, the operator must be unaffected if its argument relations is extended by an additional attribute, i.e., the operator must be *schema robust*.

Definition 2. (*Schema Robust Operator*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be relations, where relation $\mathbf{r}_i$ has schema $R_i = (\mathbf{A}_i)$, and ψ be an n-ary operator that yields a relation with schema $\mathbf{E}$ when applied to $\mathbf{r}_1, \ldots, \mathbf{r}_n$. Let $\mathbf{r}'_1, \ldots \mathbf{r}'_n$ be relations where $\mathbf{r}'_i$ has schema $R'_i = (\mathbf{A}_i, \mathbf{X}_i)$ and let $\mathbf{r}_i \equiv \pi_{\mathbf{A}_i}(\mathbf{r}'_i)$.

Operator ψ is schema robust iff for all $\mathbf{X}_i$ and $\mathbf{r}_1, \ldots, \mathbf{r}_n$ the following holds:

$$\psi(\mathbf{r}_1, \ldots, \mathbf{r}_n) \equiv \pi_{\mathbf{E}}(\psi(\mathbf{r}'_1, \ldots, \mathbf{r}'_n)).$$

Definition 3. (*Extend Operator*) Let $\mathbf{r}$ be a temporal relation with schema $(A_1, \ldots, A_m, T)$. The *extend operator*, $\epsilon_U(\mathbf{r})$, yields a temporal relation with schema $(A_1, \ldots, A_m, U, T)$ and is defined as follows:

$$z \in \epsilon_U(\mathbf{r}) \iff \exists r \in \mathbf{r}(z.\mathbf{A} = r.\mathbf{A} \wedge z.U = r.T \wedge z.T = r.T).$$

Definition 4. (*Extended Snapshot Reducibility*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be temporal relations, ψ^T an n-ary schema robust temporal operator, and ψ the corresponding n-ary nontemporal operator that yields a relation with schema $\mathbf{E}$. Let Ω^T be the time domain and $\tau_p(\mathbf{r})$ be the timeslice operator. Operator ψ^T is *extended snapshot reducible* to ψ iff

$$\begin{aligned}\forall t \in \Omega^T(&\tau_t(\psi^T(\mathbf{r}_1, \ldots, \mathbf{r}_n)) \\ &\equiv \pi_{\mathbf{E}}(\psi(\tau_t(\epsilon_{U_1}(\mathbf{r}_1)), \ldots, \tau_t(\epsilon_{U_n}(\mathbf{r}_n))))),\end{aligned}$$

where in predicates and functions on the right-hand side $\mathbf{r}_i.T$ has been substituted with U_i.

The crucial property of extended snapshot reducibility is that it allows references to timestamps by substituting them with references to explicit attributes that have been propagated for this purpose.

Example 5. Consider our running example in Fig. 2.1. We illustrate extended snapshot reducibility for timepoint $2012/1$ and query $Q1 = \mathbf{R} \rtimes^T_{Min \leq DUR(\mathbf{R}.T) \leq Max} \mathbf{P}$.

1. Propagate the timestamp of $\mathbf{R}$ by extending relation $\mathbf{R}$:

$\epsilon_U(\mathbf{R}) =$

	N	U	T
r_1	Ann	$[2012/1, 2012/8)$	$[2012/1, 2012/8)$
r_2	Joe	$[2012/2, 2012/6)$	$[2012/2, 2012/6)$
r_3	Ann	$[2012/8, 2012/12)$	$[2012/8, 2012/12)$

2. Determine snapshots at timepoint $2012/1$:

$$\tau_{2012/1}(\epsilon_U(\mathbf{R})) = \{(\text{Ann}, [2012/1, 2012/8))\},$$
$$\tau_{2012/1}(\mathbf{P}) = \{(50, 1, 2), (40, 3, 7), (30, 8, 12)\}.$$

3. Substitute $\mathbf{R}.T$ in the condition of the left outer join with U and compute a nontemporal left outer join:

$$\tau_{2012/1}(\epsilon_U(\mathbf{R})) \,⟕_{Min \leq DUR(U) \leq Max}\, \tau_{2012/1}(\mathbf{P}) = \{(\text{Ann}, 40, 3, 7, [2012/1, 2012/8))\}.$$

4. Project on (N, A, Min, Max): $\{(\text{Ann}, 40, 3, 7)\}$.

For the construction of relational algebra expressions it is also relevant if an operator is *schema propagating* or not. For instance, all types of joins are schema robust as well as schema propagating. Temporal aggregation is schema robust but not timestamp propagating since a single result tuple is not derived from a fixed number of argument tuples.

Definition 5. (*Timestamp Propagating Operator*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be relations where relation $\mathbf{r}_i$ has schema $R_i = (\mathbf{A}_i)$, ψ an n-ary schema robust operator that yields a relation with schema $\mathbf{E}$ when applied to $\mathbf{r}_1, \ldots, \mathbf{r}_n$, and $\mathbf{r}'_1, \ldots \mathbf{r}'_n$ relations where $\mathbf{r}'_i$ has schema $R'_i = (\mathbf{A}_i, \mathbf{X}_i)$. Operator ψ is *timestamp propagating* iff

$$sch(\psi(\mathbf{r}_1, \ldots, \mathbf{r}_n)) = (\mathbf{E})$$
$$\Rightarrow sch(\psi(\mathbf{r}'_1, \ldots, \mathbf{r}'_n)) = (\mathbf{E}, \mathbf{X}_1, \ldots, \mathbf{X}_n)$$

Table 2.1 summarizes schema robust and timestamp propagating operators, respectively.

Operators	Schema robust	Timestamp propagating
$\{\sigma, \times, \bowtie, ⟕, ⟖, ⟗, \triangleright\}$	yes	yes
$\{\pi, \vartheta\}$	yes	no
$\{-, \cap, \cup\}$	no	no

Table 2.1: Properties of Operators.

2.3.4 Change Preservation

Data lineage [CWW00; BT09] traces how result tuples are derived from argument tuples and has been studied in contexts where the relationship between argument and result tuples is relevant. We show that data lineage nicely complements snapshot reducibility and can be used to define a natural and unique grouping of time points into intervals that is change preserving.[1] For instance, given the result of query $Q1$ in Fig. 2.1b between $2012/6$ and $2012/9$, (extended) snapshot reducibility only defines that at each of these timepoints a tuple with values $(\text{Ann}, \omega, \omega, \omega)$ must exist. That means that replacing tuples z_3 and z_4 by a single tuple $z_{34} = (\text{Ann}, \omega, \omega, \omega, [2012/6, 2012/10))$, or four tuples over one month each would not violate (extended) snapshot reducibility.

Definition 6. (*Lineage Set*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be temporal relations and $z \in \psi^T(\mathbf{r}_1, \ldots, \mathbf{r}_n)$ be a result tuple at timepoint t of an n-ary (extended) snapshot reducible temporal operator ψ^T. The *lineage set*, $\mathsf{L}[\psi^T(\mathbf{r}_1, \ldots, \mathbf{r}_n)](z, t)$, of result tuple z at time point t is the list of sets of argument tuples, $\langle \mathbf{r}'_1, \ldots, \mathbf{r}'_n \rangle$, $\mathbf{r}'_i \subseteq \mathbf{r}_i$ from which z is derived:

$$
\begin{aligned}
\mathsf{L}[\sigma^T_\theta(\mathbf{r})](z,t) &= \langle \{r \in \mathbf{r} \mid z.\mathbf{A} = r.\mathbf{A} \wedge \theta(r) \wedge t \in r.T\} \rangle \\
\mathsf{L}[\pi^T_{\mathbf{B}}(\mathbf{r})](z,t) &= \langle \{r \in \mathbf{r} \mid z.\mathbf{B} = r.\mathbf{B} \wedge t \in r.T\} \rangle \\
\mathsf{L}[\mathbf{r} -^T \mathbf{s}](z,t) &= \langle \{r \in \mathbf{r} \mid z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}, \mathbf{s} \rangle \\
\mathsf{L}[\mathbf{r} \cup^T \mathbf{s}](z,t) &= \langle \{r \in \mathbf{r} \mid z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}, \\
&\qquad \{s \in \mathbf{s} \mid z.\mathbf{A} = s.\mathbf{C} \wedge t \in s.T\} \rangle \\
\mathsf{L}[\mathbf{r} \times^T \mathbf{s}](z,t) &= \langle \{r \in \mathbf{r} \mid z.\mathbf{A} = r.\mathbf{A} \wedge t \in r.T\}, \\
&\qquad \{s \in \mathbf{s} \mid z.\mathbf{C} = s.\mathbf{C} \wedge t \in s.T\} \rangle \\
\mathsf{L}[\mathbf{r} \rtimes^T_\theta \mathbf{s}](z,t) &= \begin{cases} \mathsf{L}[\mathbf{r} \rhd^T_\theta \mathbf{s}](z,t) & \text{if } z.\mathbf{C} = (\omega, \ldots, \omega) \\ \mathsf{L}[\mathbf{r} \bowtie^T_\theta \mathbf{s}](z,t) & \text{otherwise} \end{cases} \\
\mathsf{L}[\mathbf{r} \ltimes^T_\theta \mathbf{s}](z,t) &= \begin{cases} \mathsf{L}[\mathbf{s} \rhd^T_\theta \mathbf{r}](z,t) & \text{if } z.\mathbf{A} = (\omega, \ldots, \omega) \\ \mathsf{L}[\mathbf{r} \bowtie^T_\theta \mathbf{s}](z,t) & \text{otherwise} \end{cases}
\end{aligned}
$$

[1] Originally, when introduced in the context of the sequenced semantics [BJS00; BJ09], this property was termed interval preservation. We use the term change preservation, which better captures its nature.

$$\mathsf{L}[\mathbf{r} \,\tilde{\bowtie}^T_\theta \mathbf{s}](z,t) = \begin{cases} \mathsf{L}[\mathbf{s} \triangleright^T_\theta \mathbf{r}](z,t) & \text{if } z.\mathbf{A} = (\omega,\ldots,\omega) \\ \mathsf{L}[\mathbf{r} \triangleright^T_\theta \mathbf{s}](z,t) & \text{if } z.\mathbf{C} = (\omega,\ldots,\omega) \\ \mathsf{L}[\mathbf{r} \bowtie^T_\theta \mathbf{s}](z,t) & \text{otherwise} \end{cases}$$

Example 6. Consider the temporal left outer join in Example 4 with the result from Fig. 2.1b:

- $\mathsf{L}[\mathbf{R} \,\tilde{\bowtie}^T_\theta \mathbf{P}](z_1, 2012/2) = \langle\{r_1\}, \{s_2\}\rangle$,
- $\mathsf{L}[\mathbf{R} \,\tilde{\bowtie}^T_\theta \mathbf{P}](z_3, 2012/6) = \langle\{r_1\}, \{s_1, s_2, s_3, s_4, s_5\}\rangle$.

Note that lineage sets for inner join, aggregation, intersection and antijoin are not listed explicitly in Def. 6 since they are identical to, respectively, Cartesian product, projection, union and difference (e.g., $\mathsf{L}[\mathbf{r} \bowtie^T_\theta \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} \times^T \mathbf{s}](z,t)$). The definitions are identical since the specifics of the operators, e.g., theta conditions, are part of the definition of the result tuples z. The result tuples are defined through (extended) snapshot reducibility, which includes the theta conditions, and the result tuples are arguments in the definition of the lineage set. In the following we omit the operator and write $\mathsf{L}(z,t)$ if we discuss general properties of lineage sets.

Similar to nontemporal operators [CWW00], the lineage set for temporal operators has three properties: (i) when an operator is applied to the lineage set, $\mathsf{L}(z,t)$, its result at time t is identical to the snapshot of tuple z at time t, (ii) each tuple in the lineage set contributes to the result tuple and (iii) the lineage set is maximal.

Lineage sets trace the result tuples at a time point back to the argument tuples. Together with (extended) snapshot reducibility they define the result of a temporal operator. By merging contiguous time points with identical lineage sets we obtain result tuples over maximal time intervals that preserve changes.

Definition 7. (*Change Preservation*) Let $\mathbf{r}_1, \ldots, \mathbf{r}_n$ be temporal relations, $\mathbf{z} = \psi^T(\mathbf{r}_1, \ldots, \mathbf{r}_n)$ be the result of an n-ary temporal operator ψ^T, and let $\mathsf{L}(z,p)$ be the lineage set of result tuple $z \in \mathbf{z}$ at timepoint t. The temporal operator, ψ^T, is *change preserving* iff for all $z \in \mathbf{z}$ and $z' \in \mathbf{z}$ the following holds:

$$\begin{aligned} &\forall t, t' \in z.T(\mathsf{L}(z,t) = \mathsf{L}(z,t')) \wedge \\ &(z.T_S - 1 \in z'.T \wedge z.\mathbf{A} = z'.\mathbf{A} \Rightarrow \mathsf{L}(z', z.T_S - 1) \neq \mathsf{L}(z, z.T_S)) \wedge \\ &(z.T_E \in z'.T \wedge z.\mathbf{A} = z'.\mathbf{A} \Rightarrow \mathsf{L}(z', z.T_E) \neq \mathsf{L}(z, z.T_S)). \end{aligned}$$

Example 7. Consider the temporal left outer join in Fig. 2.1b. For result tuples z_3 and z_4 we have the following lineage sets:

- $\forall p \in z_3.T$: $\mathsf{L}[\mathbf{R} \overset{T}{\mathbb{T}}\!\!\bowtie_\theta \mathbf{P}](z_3, p) = \langle\{r_1\}, \mathbf{P}\rangle$
- $\forall p' \in z_4.T$: $\mathsf{L}[\mathbf{R} \overset{T}{\mathbb{T}}\!\!\bowtie_\theta \mathbf{P}](z_4, p') = \langle\{r_3\}, \mathbf{P}\rangle$

The change at time $2012/8$ where one reservation of Ann ends and a different reservation of Ann starts is preserved. Any result relation with more tuples over smaller time intervals would not preserve changes. For example, replacing z_3 in Fig. 2.1b by $z_3' = (\text{Ann}, \omega, \omega, \omega, [2012/6, 2012/7))$ and $z_3'' = (\text{Ann}, \omega, \omega, \omega, [2012/7, 2012/8))$ violates the maximality constraint since their lineage sets would be equal and the two tuples are value-equivalent and adjacent.

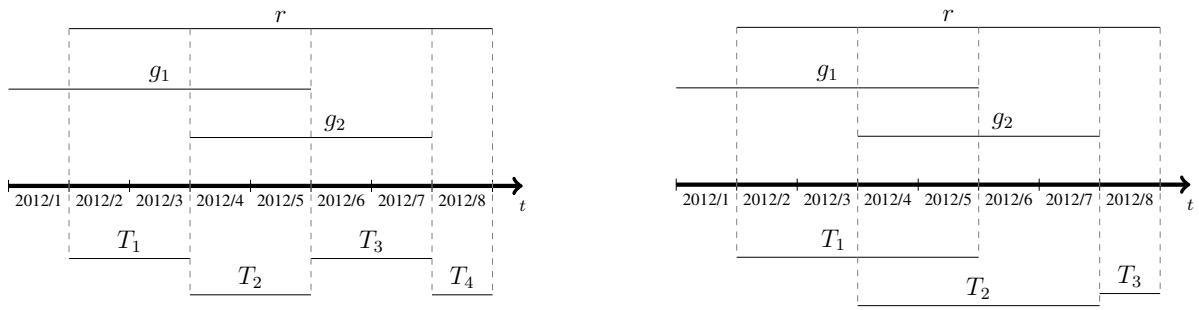
2.4 Temporal Primitives

This section introduces two temporal primitives that will be used in Sec. 2.5 to reduce the operators of a temporal algebra to operators of the nontemporal relational algebra, while preserving the three properties of the sequenced semantics.

Based on the characteristics of how operators produce result tuples, we identify group and tuple based operators. *Group based* operators are $\{\pi, \vartheta, \cup, -, \cap\}$. All tuples with identical values for the (grouping) attributes contribute to one result tuple. *Tuple based* operators are $\{\sigma, \times, \bowtie, \overset{T}{\mathbb{T}}\!\!\bowtie, \bowtie\!\!\!\!\!\!\subset, \overset{T}{\mathbb{T}}\!\!\bowtie\!\!\!\!\!\!\subset, \triangleright\}$. For these operators at most one tuple of every argument relation contributes to the values of a single result tuple. For each operator class we design a temporal primitive that provides both equality on the adjusted timestamps and change preservation for the subsequent nontemporal operation.

2.4.1 Temporal Splitter

For group based operators, $\{\pi, \vartheta, \cup, -, \cap\}$, we propose a temporal splitter primitive that adjusts the time interval of an argument tuple by splitting it at each start and end point of all tuples in the same group.



(a) Temporal Splitter.

(b) Temporal Aligner.

Figure 2.2: Temporal Splitter and Aligner.

Definition 8. *(Temporal Splitter)* Let r be a tuple and $\mathbf{g}$ a set of tuples. A *temporal splitter* produces a set of tuples with the nontemporal attributes of r over the following adjusted intervals:

$$
\begin{aligned}
&T \in \mathit{split}(r, \mathbf{g}) \iff \\
&\quad T \subseteq r.T \wedge \forall g \in \mathbf{g}(g.T \cap T = \emptyset \vee T \subseteq g.T) \wedge \\
&\quad \forall T' \supset T(\exists g \in \mathbf{g}(T' \cap g.T \neq \emptyset \wedge T' \not\subseteq g.T) \vee T' \not\subseteq r.T).
\end{aligned}
$$

The second line requires that an adjusted interval, T, is contained in r's timestamp and either contained or disjoint from all timestamp intervals of tuples $g \in \mathbf{g}$. The third line requires that T is maximal, i.e., it cannot be enlarged without violating the first condition.

Example 8. Figure 2.2a illustrates the temporal splitter with $\mathbf{g} = \{g_1, g_2\}$. The temporal splitter produces maximal sub-intervals of r's timestamp that are contained in the intervals of all overlapping tuples.

A temporal primitive that satisfies the properties of a temporal splitter is the normalization function of Toman [Tom97].

Definition 9. (*Normalization [Tom97, Sec. 4]*) Let $\mathbf{r}$ be a temporal relation. The *normalization*, $\mathcal{N}_{\mathbf{B}}(\mathbf{r}, \mathbf{s})$, of $\mathbf{r}$ with respect to $\mathbf{s}$ and attributes $\mathbf{B} \subseteq \mathbf{r}.\mathbf{A}$ is defined as follows:

$$
\begin{aligned}
&\tilde{r} \in \mathcal{N}_{\mathbf{B}}(\mathbf{r}, \mathbf{s}) \iff \\
&\quad \exists r \in \mathbf{r}(\tilde{r}.\mathbf{A} = r.\mathbf{A} \wedge \tilde{r}.T \in \mathit{split}(r, \{s \in \mathbf{s} \mid s.\mathbf{B} = r.\mathbf{B}\})).
\end{aligned}
$$

Proposition 1. *Assume a temporal relation* $\mathbf{r}$ *and the temporal normalization* $\tilde{\mathbf{r}} = \mathcal{N}_{\mathbf{B}}(\mathbf{r}, \mathbf{r})$. *All tuples* $\tilde{r} \in \tilde{\mathbf{r}}$ *with the same* $\mathbf{B}$*-values have interval timestamps that are either equal or disjoint.*

Proposition 2. *Assume temporal relations* $\mathbf{r}$ *and* $\mathbf{s}$ *with schema* $(\mathbf{A}, T)$ *and the temporal normalizations* $\tilde{\mathbf{r}} = \mathcal{N}_{\mathbf{A}}(\mathbf{r}, \mathbf{s})$ *and* $\tilde{\mathbf{s}} = \mathcal{N}_{\mathbf{A}}(\mathbf{s}, \mathbf{r})$*. Any two tuples* $\tilde{r} \in \tilde{\mathbf{r}}$ *and* $\tilde{s} \in \tilde{\mathbf{s}}$ *with the same* $\mathbf{A}$*-values have interval timestamps that are either equal or disjoint.*

Example 9. Figure 2.3 illustrates the temporal normalization $\mathcal{N}_{\{\}}(\mathbf{R}, \mathbf{R})$ for relation $\mathbf{R}$ from Example 4. For instance, tuple $(\text{Ann}, [2012/2, 2012/6))$ is derived from r_1 over a maximal subinterval that is either identical or disjoint from the intervals of all other result tuples.

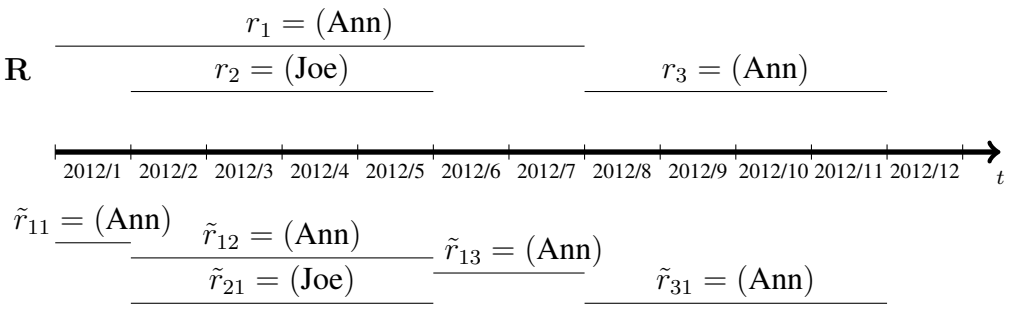


Figure 2.3: Temporal Normalization.

2.4.2 Temporal Aligner

For tuple based operators, $\{\sigma, \times, \bowtie, ⟕, ⟖, ⟗, \triangleright\}$, we propose a temporal aligner primitive that adjusts an argument tuple according to each tuple of a group.

Definition 10. *(Temporal Aligner)* Let r be a tuple and $\mathbf{g}$ be a set of tuples. A *temporal aligner* produces a set of tuples with the nontemporal attributes of r over the following adjusted intervals:

$$
\begin{aligned}
&T \in \mathit{align}(r, \mathbf{g}) \iff \\
&\quad \exists g \in \mathbf{g}(T = r.T \cap g.T) \wedge T \neq \emptyset \vee \\
&\quad T \subseteq r.T \wedge \forall g \in \mathbf{g}(g.T \cap T = \emptyset) \wedge \\
&\quad \forall T' \supset T(\exists g \in \mathbf{g}(T' \cap g.T \neq \emptyset) \vee T' \not\subseteq r.T).
\end{aligned}
$$

The second line handles all possible sub-intervals of $r.T$ for which a timestamp interval in $\mathbf{g}$ exists: in this case T is their intersection. The third and fourth lines handle sub-intervals for which no covering interval in $\mathbf{g}$ exists: in this case T is a maximal non-covered part of $r.T$.

Example 10. Figure 2.2b illustrates the temporal aligner with $\mathbf{g} = \{g_1, g_2\}$. The time intervals T_1 and T_2 are derived from the intersection of r with g_1 and g_2, respectively. T_3 is a sub-interval of $r.T$ that is not covered by any tuple in $\mathbf{g}$.

Definition 11. (*Temporal Alignment*) Let $\mathbf{r}$ and $\mathbf{s}$ be two temporal relations and θ be a predicate over the nontemporal attributes of a tuple in $\mathbf{r}$ and a tuple in $\mathbf{s}$. The *temporal alignment* operator, $\mathbf{r}\Phi_\theta\mathbf{s}$, of $\mathbf{r}$ with respect to $\mathbf{s}$ and condition θ is defined as follows:

$$\tilde{r} \in \mathbf{r}\Phi_\theta\mathbf{s} \iff \exists r \in \mathbf{r}(\tilde{r}.A = r.A \wedge \tilde{r}.T \in align(r, \{s \in \mathbf{s} \mid \theta(r,s)\})).$$

Example 11. Figure 2.4 shows the alignment of $\mathbf{P}$ with respect to $\epsilon_U(\mathbf{R})$ using condition $\theta \equiv (Min \leq DUR(U) \leq Max)$. For instance, the first result tuple, $(50, 1, 2, [2012/1, 2012/6))$, is derived from s_1 over the interval $[2012/1, 2012/6)$ for which no tuple in the other relation exists that satisfies θ. The second result tuple, $(40, 3, 7, [2012/1, 2012/6))$, is derived from s_2 and r_1 over their common interval, and the third result tuple, $(40, 3, 7, [2012/2, 2012/6))$, from s_2 and r_2 over their common interval. Notice that the second and third tuple are value-equivalent over overlapping timepoints and are both derived from tuple s_2.

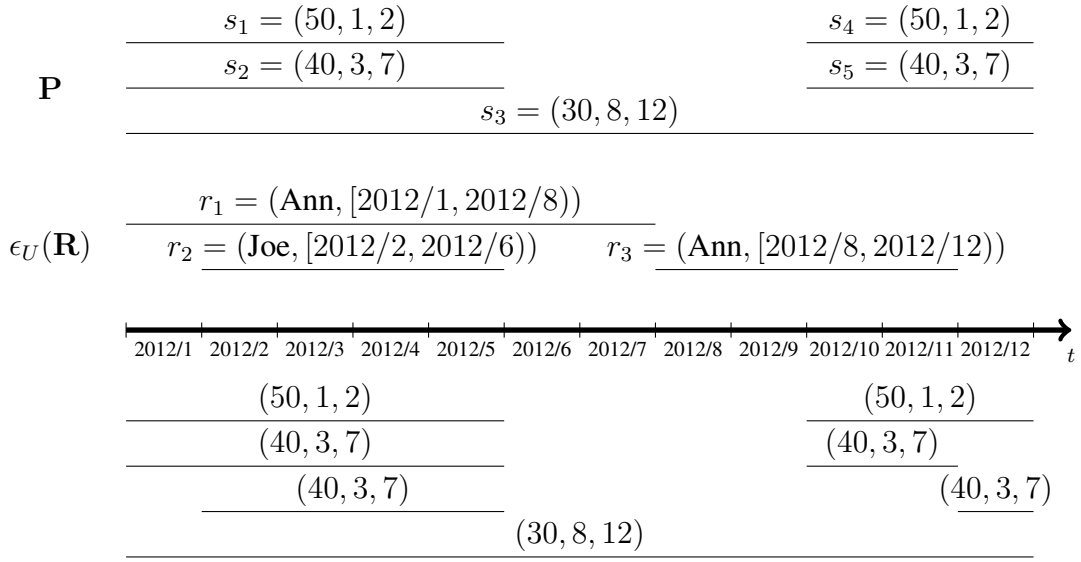


Figure 2.4: Temporal Alignment.

Lemma 1. *Let* $\mathbf{r}$ *be a temporal relation with* $|\mathbf{r}| = n$, $\mathbf{s}$ *be a temporal relation with* $|\mathbf{s}| = m$, *and* $\tilde{\mathbf{r}} = \mathbf{r}\Phi_\theta\mathbf{s}$ *be the result of temporal alignment with condition* θ. *The upper bound of the cardinality of the aligned relation is* $|\tilde{\mathbf{r}}| \leq 2nm + n$.

Proof. By induction. Base case: $n = 1$. The result of unifying a relation $\mathbf{r} = \{r_1\}$ with a relation $\mathbf{s} = \{s_1, \ldots, s_m\}$ generates at most $2m + 1$ result tuples. There exist at most m sub-intervals of $r_1.T$ that overlap with a tuple in $\mathbf{s}$ and at most $m + 1$ sub-intervals of $r_1.T$ that do not overlap with any tuple in $\mathbf{s}$. This is illustrated in Fig. 2.5 for $n = 1$ and $m = 2$, where r_1 is split into $2 * 2 + 1 = 5$ result tuples. Inductive case: $n > 1$. Assume an argument relation $\mathbf{r}$ with n tuples that can have up to $2nm + n$ output tuples. Then $n + 1$ tuples in the argument relation can produce $2(n + 1)m + (n + 1)$ tuples. This holds since $2mn + n$ tuples can be produced by n argument tuples and an additional tuple can yield up to $2m + 1$ new result tuples. □

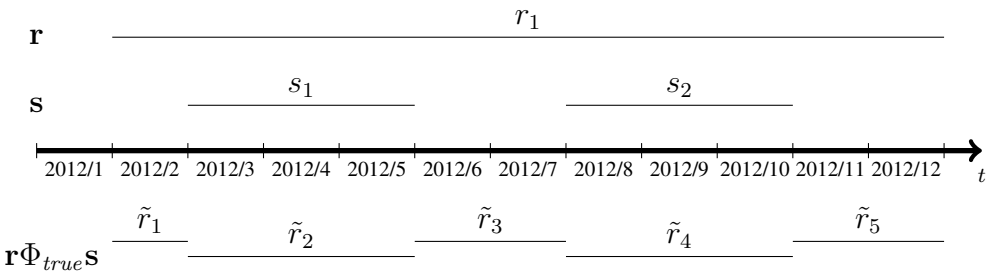


Figure 2.5: Base Case (for $n = 1$ and $m = 2$).

Proposition 3. *Assume temporal relations* $\mathbf{r}$ *and* $\mathbf{s}$ *with alignments* $\tilde{\mathbf{r}} = \mathbf{r}\Phi_\theta\mathbf{s}$ *and* $\tilde{\mathbf{s}} = \mathbf{s}\Phi_\theta\mathbf{r}$. *For any two tuples* $r \in \mathbf{r}$ *and* $s \in \mathbf{s}$ *that satisfy* θ *and* $r.T \cap s.T \neq \emptyset$, *there are two tuples* $\tilde{r} \in \tilde{\mathbf{r}}$ *and* $\tilde{s} \in \tilde{\mathbf{s}}$ *with matching nontemporal values for* r *and* s, *respectively, and with identical timestamps* $\tilde{r}.T = \tilde{s}.T = r.T \cap s.T$.

Proposition 4. *Assume temporal relations* $\mathbf{r}$ *and* $\mathbf{s}$. *Every tuple* $\tilde{r} \in \mathbf{r}\Phi_\theta\mathbf{s}$ *is derived from a tuple* $r \in \mathbf{r}$, *and the timestamp of* $\tilde{r}$ *is either the intersection of* $r.T$ *with the timestamp of a tuple* $s \in \mathbf{s}$ *satisfying* θ, *or a maximal sub-interval of* $r.T$ *that is not covered by the interval timestamp of a tuple* $s \in \mathbf{s}$ *satisfying* θ.

2.5 Reducing Temporal Operators

This section uses temporal splitter and aligner to reduce operators with sequenced semantics to their nontemporal counterparts.

2.5.1 Overview

Figure 2.6 illustrates the basic scheme for reducing temporal operators with sequenced semantics. We assume extended relations (cf. Sec. 2.3.3). Thus, all references to timestamps have been substituted with references to explicit attributes with propagated timestamps.

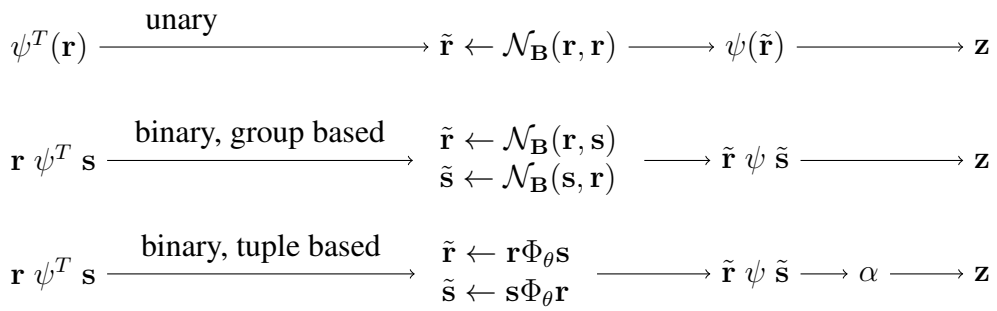


Figure 2.6: Reduction of Temporal Operators.

The normalization or alignment primitive transforms argument relation(s) with overlapping timestamps into temporal relations where the timestamps of tuples have been adjusted. Only equality is required to compare such timestamps. This allows to replace the temporal operator by the corresponding nontemporal operator on adjusted relations and an equality on the timestamps.

Before giving the reduction rules we need a final operator to eliminate temporal duplicates. The alignment primitive produces all distinct intersections of matching tuples for tuple based operators. Since the timestamps are adjusted independently for each tuple, the result might include intervals that are not maximal intersections of two tuples as illustrated in the next example.

Example 12. Consider the Cartesian product of relations $\mathbf{r} = \{(a,[1,9)),(b,[3,7))\}$ and $\mathbf{s} = \{(c,[1,9)),(d,[3,7))\}$. The temporal alignment produces $\tilde{\mathbf{r}} = \Phi_{true}(\mathbf{r},\mathbf{s}) = \{(a,[1,9)),$ $(a,[3,7)),(b,[3,7))\}$. Similar for $\mathbf{s}$ we get $\tilde{\mathbf{s}} = \{(c,[1,9)),(c,[3,7)),(d,[3,7))\}$. The subsequent equality join of $\tilde{\mathbf{r}}$ and $\tilde{\mathbf{s}}$ on the adjusted timestamp attributes (cf. reduction rule for the Cartesian product in Table 2.2) gives:

z_1	a	c	$[1,9)$
z_2	a	c	$[3,7)$
z_3	a	d	$[3,7)$
z_4	b	c	$[3,7)$
z_5	b	d	$[3,7)$

Tuple z_2 is produced by joining $\tilde{r}_2 = (a, [3, 7))$ and $\tilde{s}_2 = (c, [3, 7))$ and is a temporal duplicate of z_1. Note that we cannot remove $\tilde{r}_2$ or $\tilde{s}_2$ before the join, since these tuples are required to produce tuples z_3 and z_4, respectively. Instead, the absorb operator removes temporal duplicates in a post-processing step.

Definition 12. (*Absorb Operator*) Let $\mathbf{r}$ be a temporal relation with timestamp attribute T. The *absorb* operator, α, eliminates all tuples $r \in \mathbf{r}$ for which another value-equivalent tuple $r' \in \mathbf{r}$ exists such that $r.T \subset r'.T$:

$$\alpha(\mathbf{r}) = \{r \in \mathbf{r} \mid \nexists r' \in \mathbf{r}(r.\mathbf{A} = r'.\mathbf{A} \wedge r.T \subset r'.T)\}.$$

2.5.2 Reduction Rules

The following theorem defines the reduction rules for a temporal algebra with sequenced semantics.

Theorem 1. *Let* $\mathbf{r}$ *and* $\mathbf{s}$ *be temporal relations,* θ *be a predicate,* F *be a set of aggregation functions over* $\mathbf{r}.\mathbf{A}$*,* $\mathbf{B} \subseteq \mathbf{A}$ *be a set of attributes and* α *be the absorb operator. The reduction rules in Table 2.2 define a temporal algebra with sequenced semantics.*

Operator			**Reduction**
Selection	$\sigma^T_\theta(\mathbf{r})$	=	$\sigma_\theta(\mathbf{r})$
Projection	$\pi^T_{\mathbf{B}}(\mathbf{r})$	=	$\pi_{\mathbf{B},T}(\mathcal{N}_{\mathbf{B}}(\mathbf{r},\mathbf{r}))$
Aggregation	${}_{\mathbf{B}}\vartheta^T_F(\mathbf{r})$	=	${}_{\mathbf{B},T}\vartheta_F(\mathcal{N}_{\mathbf{B}}(\mathbf{r},\mathbf{r}))$
Difference	$\mathbf{r} -^T \mathbf{s}$	=	$\mathcal{N}_{\mathbf{A}}(\mathbf{r},\mathbf{s}) - \mathcal{N}_{\mathbf{A}}(\mathbf{s},\mathbf{r})$
Union	$\mathbf{r} \cup^T \mathbf{s}$	=	$\mathcal{N}_{\mathbf{A}}(\mathbf{r},\mathbf{s}) \cup \mathcal{N}_{\mathbf{A}}(\mathbf{s},\mathbf{r})$
Intersection	$\mathbf{r} \cap^T \mathbf{s}$	=	$\mathcal{N}_{\mathbf{A}}(\mathbf{r},\mathbf{s}) \cap \mathcal{N}_{\mathbf{A}}(\mathbf{s},\mathbf{r})$
Cart. Prod.	$\mathbf{r} \times^T \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_{true}\mathbf{s}) \bowtie_{\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_{true}\mathbf{r}))$
Inner Join	$\mathbf{r} \bowtie^T_\theta \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_\theta\mathbf{s}) \bowtie_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r}))$
Left O. Join	$\mathbf{r} ⟕^T_\theta \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_\theta\mathbf{s}) ⟕_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r}))$
Right O. Join	$\mathbf{r} ⟖^T_\theta \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_\theta\mathbf{s}) ⟖_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r}))$
Full O. Join	$\mathbf{r} ⟗^T_\theta \mathbf{s}$	=	$\alpha((\mathbf{r}\Phi_\theta\mathbf{s}) ⟗_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r}))$
Anti Join	$\mathbf{r} \triangleright^T_\theta \mathbf{s}$	=	$(\mathbf{r}\Phi_\theta\mathbf{s}) \triangleright_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\mathbf{s}\Phi_\theta\mathbf{r})$

Table 2.2: Reduction Rules.

Proof. We prove the reduction rule for the temporal left outer join, $\mathbf{r} \overset{T}{\text{⟕}}_{\theta} \mathbf{s}$, by showing that the operator satisfies the three properties of the sequenced semantics.

Snapshot reducibility (cf. Def. 1): We have to show two cases. Case 1: For each pair of matching and intersecting tuples $r \in \mathbf{r}$ and $s \in \mathbf{s}$ (i.e., $\theta(r, s)$ is true and $r.T \cap s.T \neq \emptyset$) the following holds: for each $t \in r.T \cap s.T$ there exists a result tuple $z = (r.\mathbf{A}, s.\mathbf{C}, T)$ such that $t \in T$. Case 2: For each $r \in \mathbf{r}$ and interval $T' \subseteq r.T$, for which no matching and intersecting $s \in \mathbf{s}$ exists, the following holds: for each $t \in T'$ there exists a result tuple $z = (r.\mathbf{A}, \omega, \ldots, \omega, T)$ such that $t \in T$.

From Def. 10 (temporal alignment) and Proposition 4 we know that aligned tuples $\tilde{r} \in \phi_{\theta}(\mathbf{r}, \mathbf{s})$ are derived from an $r \in \mathbf{r}$ as follows: (i) for each matching and intersecting $s \in \mathbf{s}$ we get $\tilde{r} = (r.\mathbf{A}, r.T \cap s.T)$, and (ii) for each maximal subinterval $T \subseteq r.T$ that is not covered by any matching $s \in \mathbf{s}$ we get $\tilde{r} = (r.\mathbf{A}, T)$. The same holds for the aligned tuples $\tilde{s} \in \phi_{\theta}(\mathbf{s}, \mathbf{r})$.

From (i) we conclude that for any two matching and intersecting tuples $r \in \mathbf{r}$ and $s \in \mathbf{s}$, there exists an aligned tuple $\tilde{r} = (r.\mathbf{A}, r.T \cap s.T)$ and $\tilde{s} = (s.\mathbf{C}, s.T \cap r.T)$. Since intersection is commutative, $r.T \cap s.T = s.T \cap r.T$, and the nontemporal left outer join yields a result tuple $z = (r.\mathbf{A}, s.\mathbf{C}, r.T \cap s.T)$ that covers each $t \in r.T \cap s.T$ (proves case 1). From (ii) we conclude that for each $r \in \mathbf{r}$ and maximal subinterval $T \subseteq r.T$ that has no matching and intersecting $s \in \mathbf{s}$, there exists an $\tilde{r} = (r.\mathbf{A}, T)$ but no matching $\tilde{s} \in \phi_{\theta}(\mathbf{s}, \mathbf{r})$ that intersects T. Thus, the nontemporal left outer join yields a result tuple $z = (r.\mathbf{A}, \omega, \ldots, \omega, T)$ that covers each $t \in T$ (proves case 2).

The final absorb operator, α, removes tuples that are covered by a value-equivalent tuple. Thus, if a tuple z is removed, each $t \in z.T$ is covered by another value-equivalent result tuple z'.

Extended snapshot reducibility (Def. 4): To prove extended snapshot reducibility, we show that propagated timestamps do not interfere with the alignment of the argument relations and hence with the production of result tuples. Recall that relations are extended, i.e., each $r \in \mathbf{r}$ ($s \in \mathbf{s}$) has a nontemporal attribute $r.U$ ($s.U$) that is a copy of $r.T$ ($s.T$), and in θ all references to timestamps have been substituted with $r.U$ and $s.U$, respectively. Since θ is independent of the timestamp attributes, alignment and nontemporal left outer join work exactly in the same way as for snapshot reducibility.

From (i) we conclude that for any two matching and intersecting tuples $r \in \mathbf{r}$ and $s \in \mathbf{s}$, there exists an $\tilde{r} = (r.\mathbf{A}, r.U, r.T \cap s.T)$ and an $\tilde{s} = (s.\mathbf{C}, s.U, s.T \cap r.T)$ that yield a result tuple $z = (r.\mathbf{A}, r.U, s.\mathbf{C}, s.U, r.T \cap s.T)$ that covers each $t \in r.T \cap s.T$ (proves case 1). From (ii)

we conclude that for each $r \in \mathbf{r}$ and maximal sub-interval $T \subseteq r.T$ that has no matching and intersecting $s \in \mathbf{s}$, there exists an $\tilde{r} = (r.\mathbf{A}, r.U, T)$ but no matching $\tilde{s} \in \phi_\theta(\mathbf{s}, \mathbf{r})$ that intersects T. This yields a result tuple $z = (r.\mathbf{A}, r.U, \omega, \ldots, \omega, T)$ that covers each $t \in T$ (proves case 2).

Change preservation (Def. 7): From Def. 10 (temporal alignment) and Proposition 4 we know that the timestamp of each result tuple is (case 1) either an intersection of two argument tuples, $r \in \mathbf{r}$ and $s \in \mathbf{s}$, or (case 2) a maximal subinterval $T \in r.T$ for which no matching and intersecting $s \in \mathbf{s}$ exists. Furthermore, the α-operator ensures that all result tuples have maximal timestamps.

Case 1: We show that for each result tuple $z = (r.\mathbf{A}, z.\mathbf{C}, r.T \cap s.T)$, the lineage set $\mathsf{L}[\mathbf{r} \tilde{\bowtie}_\theta^T \mathbf{s}](z,t)$ is equal for each $t \in z.T$ and that adjacent value-equivalent tuples have different lineage sets. From Def. 6 (lineage sets) we get $\mathsf{L}[\mathbf{r} \tilde{\bowtie}_\theta^T \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} \bowtie_\theta^T \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} \times^T \mathbf{s}](z,t)$ for case (1). The lineage set of the temporal Cartesian product contains all $r \in \mathbf{r}$ that are value-equivalent to $z.\mathbf{A}$ and cover t and all $s \in \mathbf{s}$ that are value-equivalent to $z.\mathbf{C}$ and cover t. Since relations are duplicate free, the lineage set contains exactly one $r \in \mathbf{r}$ and one $s \in \mathbf{s}$, i.e., $\mathsf{L}[\mathbf{r} \times^T \mathbf{s}](z,t) = \langle\{r\}, \{s\}\rangle$. This holds for all $t \in r.T \cap s.T$. To show that the lineage set at timepoint $z.T_S - 1$ is different for value-equivalent tuples, recall that either $z.T_S - 1 \notin r.T$ or $z.T_S - 1 \notin s.T$ since $z.T = r.T \cap z.T$. Hence, at least one of r and s is not in the lineage set. The same reasoning applies for timepoint $z.T_E$.

Case 2: We show that for each result tuple $z = (r.\mathbf{A}, \omega, \ldots, \omega, T)$, the lineage set $\mathsf{L}[\mathbf{r} \tilde{\bowtie}_\theta^T \mathbf{s}](z,t)$ is equal for all $t \in z.T$ and that adjacent value-equivalent tuples have different lineage sets. From Def. 6 we get $\mathsf{L}[\mathbf{r} \tilde{\bowtie}_\theta^T \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} \triangleright_\theta^T \mathbf{s}](z,t) = \mathsf{L}[\mathbf{r} -^T \mathbf{s}](z,t)$. The lineage set of the temporal difference contains all $r \in \mathbf{r}$ that are value-equivalent to $z.\mathbf{A}$ and cover t as well as $\mathbf{s}$. Since relations are duplicate free, we get $\mathsf{L}[\mathbf{r} -^T \mathbf{s}](z,t) = \langle\{r\}, \mathbf{s}\rangle$. This holds for all $t \in z.T$ since $z.T = \tilde{r}.T \subseteq r.T$. To show that the lineage set of value-equivalent tuples is different at timepoint $z.T_S - 1$, recall that $z.T$ is maximal. Either $z.T_S - 1 \notin r.T$ and therefore r is not in the lineage set, or there exists a matching $s \in \mathbf{s}$ with $z.T_S - 1 \in s.T$ that would produce a join with $r.\mathbf{A}$, and thus no value-equivalent tuple to $z = (r.\mathbf{A}, \omega, \ldots, \omega, T)$ can exist. The same reasoning applies for the timepoint $z.T_E$. □

Example 13. Figure 2.7 illustrates the reduction of the temporal aggregation query $Q2 = \vartheta^T_{AVG(DUR(\mathbf{R}.T))}(\mathbf{R})$. The query determines the average duration of reservations at each timepoint. Since there is a function with a reference to a timestamp the query is governed by extended snapshot reducibility and we first extend $\mathbf{R}$ to $\epsilon_U(\mathbf{R})$ and substitute $\mathbf{R}.T$ in $Q2$ with U. Next we

normalize $\epsilon_U(\mathbf{R})$ to get tuples with timestamps that are identical or disjoint. Finally, we apply the reduced query to get the desired result.

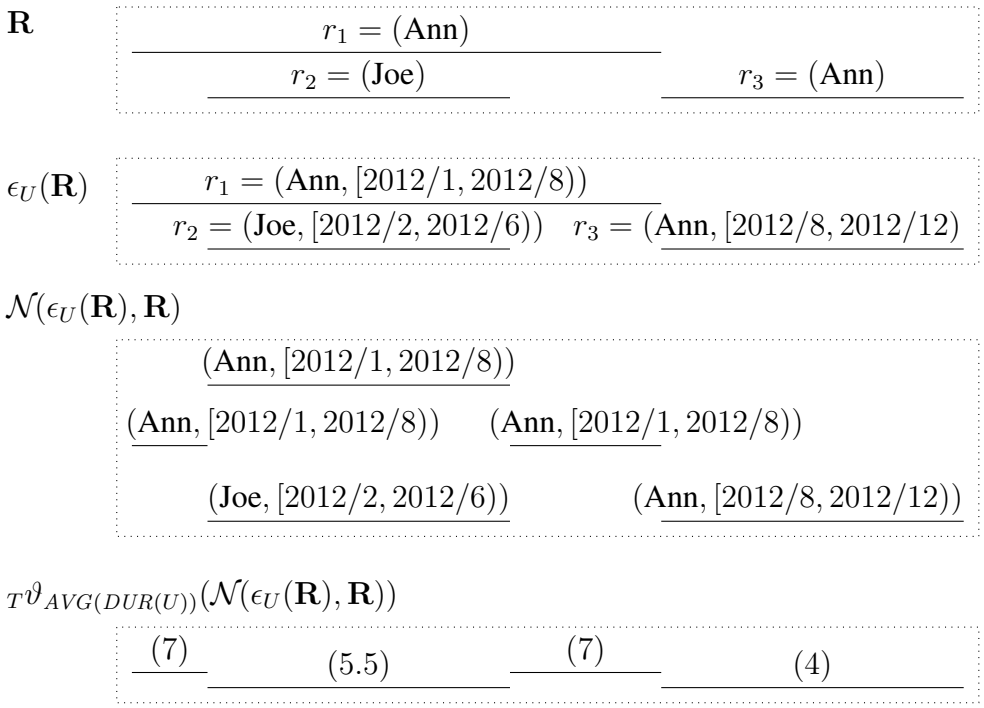


Figure 2.7: Reduction of Query Q2.

2.6 Implementation

This section describes the implementation of the temporal primitives in the kernel of the PostgreSQL database system.[2] We modified parser and parse tree, analyzer and query tree, optimizer and plan tree, and executor and execution tree. For each tree a new custom node was defined that stores information for processing the new operator. In the query processing sequence transformations between these nodes were implemented: SQL query $\overset{\text{parser}}{\longrightarrow}$ parse tree $\overset{\text{analyzer}}{\longrightarrow}$ query tree $\overset{\text{optimizer}}{\longrightarrow}$ plan tree $\overset{\text{executor}}{\longrightarrow}$ execution tree. The optimizer needs cost estimations for the new operator, and in the executor module three functions were implemented: `ExecInit`⟨`Operator`⟩, `Exec`⟨`Operator`⟩ and `ExecEnd`⟨`Operator`⟩ for initialization, execution and finalization of the evaluation algorithm, respectively, where ⟨`Operator`⟩ is the name of the actual execution algorithm.

[2] http://www.ifi.uzh.ch/dbtg/research/align.html.

To illustrate and evaluate the reduction rules, we extended SQL with the two temporal primitives. Note that this is just for illustration purposes and we do not propose a new temporal SQL. Instead, our primitives are useful building blocks that support the implementation of the temporal SQL extensions that have been proposed in the past.

2.6.1 Execution Algorithm for Temporal Alignment

The implementation of temporal alignment is a two step process: (1) we retrieve for each tuple $r_i \in \mathbf{r}$ the group $\mathbf{g}_i \subseteq \mathbf{s}$ of s-tuples that satisfy θ and (2) we apply a plane sweep algorithm on each sorted group $\mathbf{g}_i$ to produce the aligned relation.

First, we construct for each r-tuple the group $\mathbf{g}_i$ of matching s-tuples using a database internal left outer join. To illustrate our implementation, we assume two relations $\mathbf{r}$ and $\mathbf{s}$ with three tuples each and $\theta \equiv (B = D \wedge \mathbf{r}.T \cap \mathbf{s}.T \neq \emptyset)$ as illustrated in Fig. 2.8. r_1 matches two, and r_2 three s-tuples; r_3 does not match any s-tuple, hence the s-part is filled with ω values. Note that the join tuples have two timestamp attributes, from the r-tuple and the s-tuple, respectively.

r

	A	B	T
r_1	a	β	$[1,7)$
r_2	b	β	$[3,9)$
r_3	c	γ	$[8,10)$

s

	C	D	T
s_1	1	β	$[2,5)$
s_2	2	β	$[3,4)$
s_3	3	β	$[7,9)$

r ⟕$_\theta$ s

r_1	s_1
r_2	s_3
r_1	s_2
r_2	s_2
r_3	ω
r_2	s_1

Figure 2.8: Join of r-tuples with s-tuples.

Our implementation supports pipelining such that intermediate results do not have to be materialized. To make this possible the join is partitioned according to the groups and within each group sorted according to the intersection timestamp of the r and s-tuple. This ensures that tuples with equal intersection timestamps are consecutive and allows to identify (and remove) duplicate timestamps during the plane sweep. Figure 2.9 illustrates the group construction after partitioning and sorting (the sorting in each group is displayed top down; the nearby lines show the two timestamps of join tuples).

The plane sweep algorithm in Algorithm 1 is implemented in PostgreSQL as executor function `ExecAdjustment`. The function is integrated into the pipelining architecture of PostgreSQL and on each invocation either a single result tuple is returned, or ω to indicate the end of the

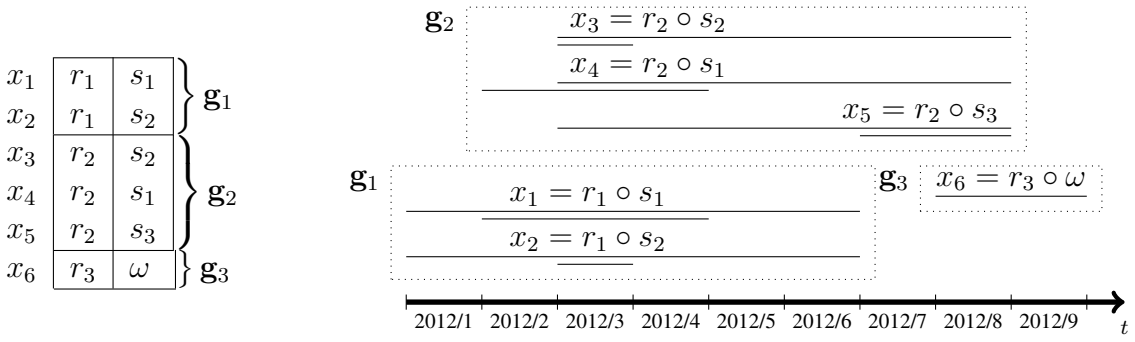


Figure 2.9: Partitioning and Sorting of Groups.

operation. The input is a context node, n, that keeps variables between consecutive invocations. Node n stores the following information: the reference to its input ($subnode$), the previous and current tuples from the input ($prev$, $curr$), the sweepline status ($sweepline$), an output tuple (out), the boolean *isalign* to distinguish alignment and normalization, and a boolean variable ($sameleft$) that is $true$ whenever $prev$ and $curr$ contain tuples from the same group and $false$ otherwise. $[P_1, P_2)$ denotes the already computed intersection of the r- and s-tuple.

Figure 2.10 illustrates four invocations of `ExecAdjustment` with four result tuples. Whenever

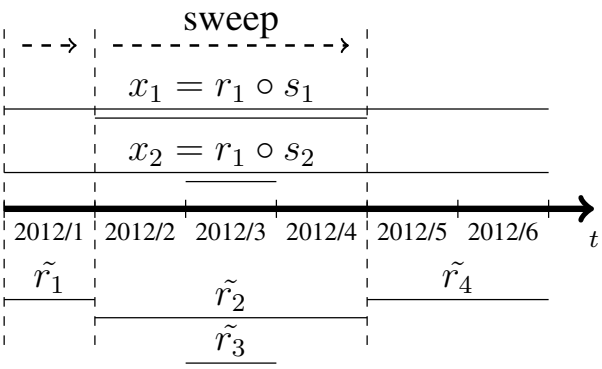


Figure 2.10: Plane Sweep Algorithm for Group $\mathbf{g}_1$.

a new group starts, $curr$ and $prev$ store the same input tuple, $sameleft$ is set to $true$ and $sweepline$ stores the r-tuple's starting timepoint. On the first invocation x_1 is fetched. $sameleft = true$ and $P_1 = 2012/2$ is larger than the $sweepline = 2012/1$. Thus, tuple $\tilde{r}_1$ is produced and the $sweepline$ is advanced to P_1 (first block of the function). On the second invocation, $sameleft = true$ and $sweepline = P_1$, hence the second block of the function is entered. We check if the same intersection has already been produced before. Since this is not the case, $\tilde{r}_2$ is produced, the sweepline is advanced to 2012/4, $curr$ is copied to $prev$, and the next tuple x_2 is fetched into

$curr$. Since x_2 belongs to the same group as x_1, $sameleft$ is set to $true$. On the third invocation, $sameleft = true$ and $sweepline > P_1$ (= 2012/3). The execution enters again in the second block and produces $\tilde{r}_3$. After updating $prev$, the next tuple x_3 is fetched into $curr$. Since x_3 belongs to a new group, $sameleft$ is set to $false$. On the fourth invocation, $sameleft = false$ and the execution enters the third block of the function. We check if $sweepline < prev.T_E$, i.e., if the timestamp of the r-tuple of the previous group is completely covered. Since this is not the case, a result tuple over the remaining part of the timestamp is produced ($\tilde{r}_4$). The variables are reset for processing the next group.

Algorithm 1: `ExecAdjustment`(n)

Input: Node n in execution tree.
Output: A single output tuple or ω.

Copy variables of n to local;
if *first call* **then**
    $prev \leftarrow curr \leftarrow$ next tuple from $subnode$;
    $sameleft \leftarrow true$;
    $sweepline \leftarrow curr.T_S$;
$produced = false$;
while $produced = false \wedge prev \neq \omega$ **do**
    if $sameleft \wedge sweepline < curr.P_1$ **then**
        $out \leftarrow (curr.\mathbf{A}, [sweepline, curr.P_1))$;
        $produced \leftarrow true$;
        $sweepline \leftarrow curr.P_1$;
    else if $sameleft \wedge sweepline \geq curr.P_1$ **then**
        if $isalign \wedge out \neq (curr.\mathbf{A}, curr.P_1, curr.P_2)$ **then**
            $out \leftarrow (curr.\mathbf{A}, [curr.P_1, curr.P_2))$;
            $sweepline \leftarrow max(sweepline, curr.P_2)$;
            $produced \leftarrow true$;
        $prev \leftarrow curr$;
        $curr \leftarrow$ next tuple from $subnode$;
        $sameleft \leftarrow prev.\mathbf{A} = curr.\mathbf{A} \wedge prev.T = curr.T$;
    else
        if $sweepline < prev.T_E$ **then**
            $out \leftarrow (prev.\mathbf{A}, [sweepline, prev.T_E))$;
            $produced \leftarrow true$;
        $prev \leftarrow curr$;
        $sweepline \leftarrow curr.T_S$;
        $sameleft \leftarrow true$;
if $produced = false$ **then**
    $out \leftarrow \omega$;
Copy local variables to n;
return out;

2.6.2 Extensions to Parser, Analyzer and Optimizer for Temporal Alignment

This section describes the extensions that are required in the three modules that precede the executor. First, we add a new SQL keyword ALIGN and extend the grammar of the parser:

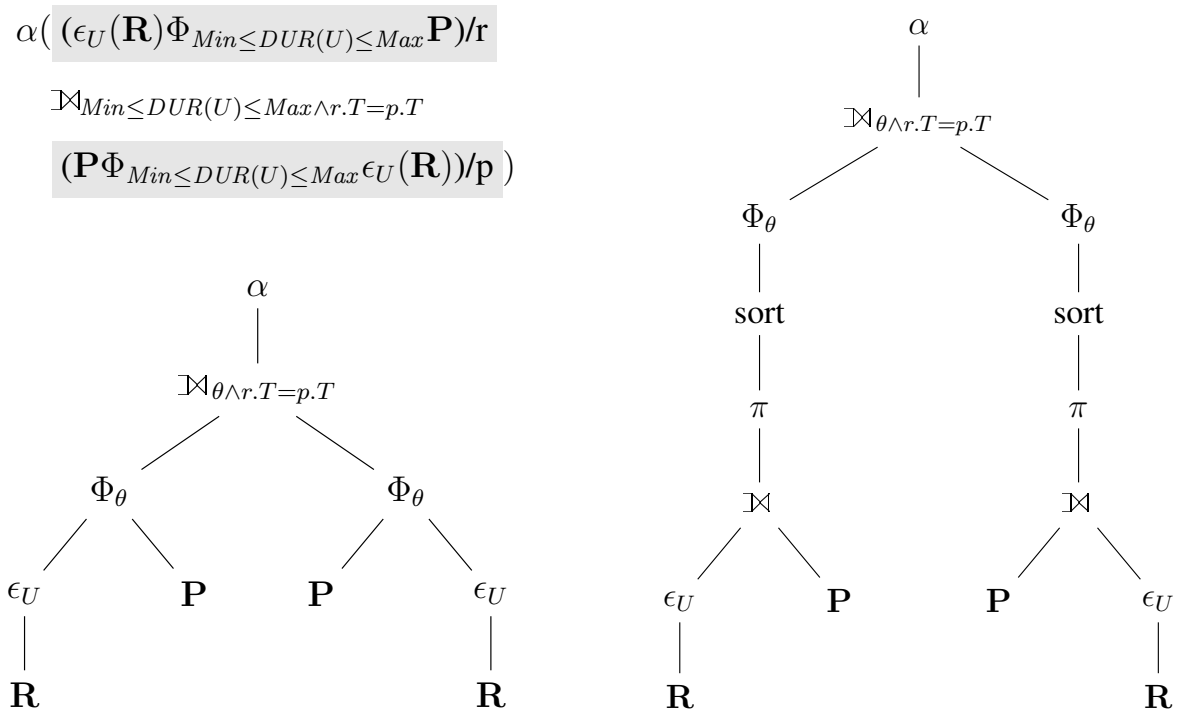
```
aligned_table:
  table_ref ALIGN table_ref ON a_expr;
table_ref: ...
  '(' aligned_table ')' alias_clause
```

The alignment statement consists of two `table_ref` and can be used similar to any other item in the FROM clause. The first `table_ref` argument is the relation to align, the second one is the reference relation; `a_expr` is the θ condition. In the select clause `ABSORB` can be specified instead of `DISTINCT` to eliminate temporal duplicates. For instance, query $Q1$ can be formulated in SQL as:

```
WITH R AS (SELECT Ts Us, Te Ue, * FROM R)
SELECT ABSORB n, a, min, max, r.Ts, r.Te
FROM (R ALIGN P ON DUR(Us,Ue) BETWEEN Min AND Max) r
     LEFT OUTER JOIN
     (P ALIGN R ON DUR(Us,Ue) BETWEEN Min AND Max) p
     ON DUR(Us,Ue) BETWEEN Min AND Max AND
        r.Ts=p.Ts AND r.Te=p.Te
```

The `WITH` statement does the timestamp propagation and the `SELECT` statement implements the reduction rule for the temporal left outer join (cf. Table 2.2). `DUR` is a user defined function (UDF) that evaluates the duration of the period defined by the two parameters. The corresponding RA expression and parse tree are shown in Fig. 2.11a.

In the analyzer we extend the query tree with the partitioning and sorting of the groups. The query tree for our example is shown in Fig. 2.11b. The optimizer is the final state before the executor. Here the database system chooses among different execution strategies. The cost estimations for



(a) Parse Tree.

(b) Query Tree.

Figure 2.11: Parse Tree and Query Tree.

our temporal alignment node, where $\mathbf{x}$ is the direct subnode, are as follows:

$$numRows = 3 * \mathbf{x}.numRows$$
$$cost = \mathbf{x}.cost + 2 * cpu\_op\_cost * \mathbf{x}.numRows * numCols$$
$$sortOrder = (\mathbf{A}, T)$$

The cardinality of the output can be up to three times the cardinality of the subnode, that is every tuple in the input can produce up to three tuples in the algorithm. The total cost is estimated by the cost of the subnode, plus two tuple comparison for each result tuple in the executor function. The result is sorted on all attributes.

2.6.3 Temporal Normalization

The approach to implement temporal normalization is similar to the implementation of temporal alignment. It differs in the construction of the groups. Temporal normalization splits a tuple's

interval according to all start and end timepoints in its group. To build the group we use a database internal nontemporal left outer join. We impose a total order on split points to get a plane sweep algorithm with constant memory complexity. Therefore we do not join with the s relation directly but with the union of its start- and endpoints, i.e., $\pi_{B,T_S/P_1}(\mathbf{s}) \cup \pi_{B,T_E/P_1}(\mathbf{s})$. We build the groups as for alignment, sort on the split point P_1, and use the same plane sweep algorithm as for temporal alignment but without the intersection part, i.e, `ExecAdjustment` with $isalign = false$. As a result the sweepline moves from split point to split point to produce the final result.

The rules for the parser are similar to temporal alignment, but we use the keyword `NORMALIZE` with a list of grouping attributes instead of a θ condition. For instance, the temporal aggregation ${}_T\vartheta_{AVG(DUR(U))}(\mathcal{N}_{\{\}}(\epsilon_U(\mathbf{R}), \epsilon_U(\mathbf{R}))\,)$ is formulated in SQL as:

```
WITH R AS (SELECT Ts Us, Te Ue, * FROM R)
SELECT AVG(DUR(Us,Ue)), Ts, Te
FROM (R R1 NORMALIZE R R2 USING()) r
GROUP BY Ts, Te
```

In the `USING` clause the grouping attributes are specified (empty in this example). In the optimizer we use the following cost estimations:

$$
\begin{aligned}
&numRows = 2 * \mathbf{x}.numRows \\
&cost = \mathbf{x}.cost + cpu\_op\_cost * \mathbf{x}.numRows * numCols \\
&sortOrder = (\mathbf{A}, T)
\end{aligned}
$$

For each split point in the subnode we can have up to two result tuples. The total cost is the cost of the subnode plus one tuple comparison for every output tuple (different from alignment since we omit the intersection part).

2.7 Empirical Evaluation

In this section we evaluate our implementation of temporal normalization and temporal alignment, by showing that (1) our implementation is tightly integrated into the database kernel and leverages existing database optimization techniques; (2) temporal normalization with change

preservation minimizes the number of splits, which keeps intermediate results small; and (3) temporal alignment remains stable for datasets that are inefficient to process with other approaches.

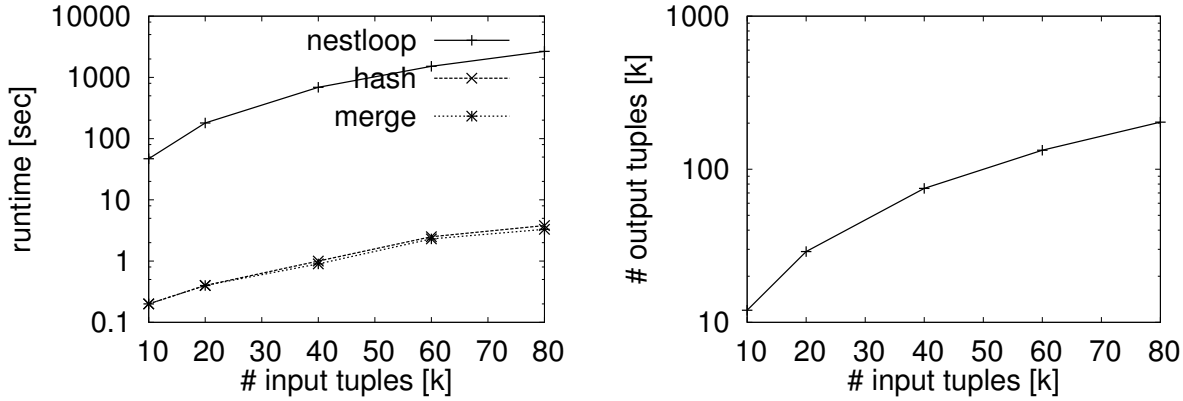
2.7.1 Setup

For the experiments the client and the database server run on the same 2.6 GHz machine with 4 GB RAM and a hard disk rotational rate of 5400 rpm. We use the PostgreSQL server 9.0, extended with our implementation of normalize and alignment. All parameters of the PostgreSQL server, such as maximum memory for sorting, were kept to default values, and no indexes were used.

We use the real world dataset $Incumbent$ [GSSY98] of the University of Arizona with 83,857 entries. Each entry records a job assignment (pcn) for an employee (ssn) over a specific time interval. The data ranges over 16 years and contains 49,195 different employees. The timestamps are recorded at the granularity of days and range from 1 to 573 days with an average of approximately 180 days. Synthetic datasets used in the evaluation are described below.

2.7.2 Database System Integration

Temporal normalization and temporal alignment fully leverage existing database optimization strategies and algorithms. The nontemporal left outer join used for the group construction in our implementation is optimized by the database system. This applies to both temporal normalization and temporal alignment. We illustrate this for temporal normalization $\mathcal{N}_{\{ssn\}}$ of the $Incumbent$ dataset, running the database in three different settings: (a) all join methods enabled, (b) merge join disabled (i.e., SET enable_mergejoin=false), and (c) merge and hash join disabled. For each of the three settings the database chooses the best suited join strategy for the left outer join in the normalization operator: in (a) a sorted merge join, in (b) a hash join, and in (c) a nested loop join is used. Figure 2.12a shows the runtime of the normalization, which is dominated by the join, for which the DBMS chooses the best available join algorithm. The same holds for temporal alignment. Hence, the runtime of normalization and alignment is proportional to the runtime of a join. The output cardinality of the normalization is shown in Fig. 2.12b, which is obviously the same in all settings.



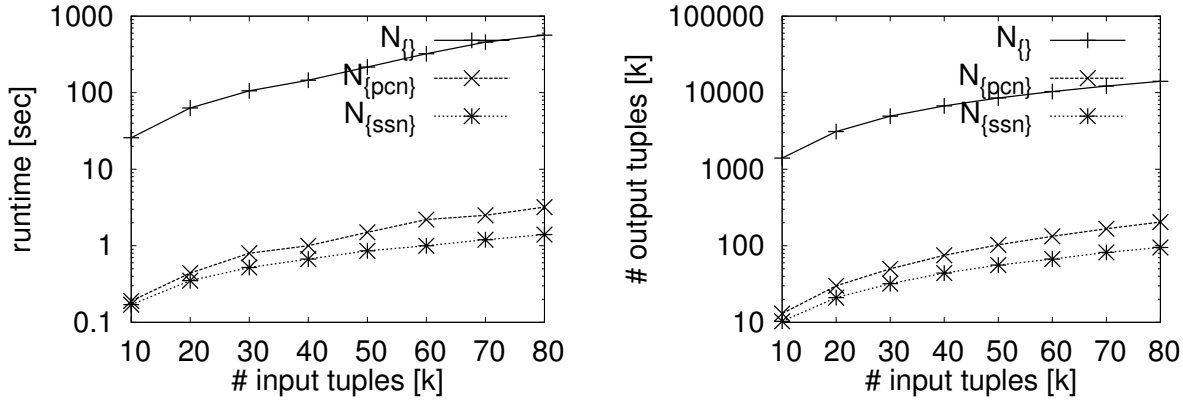
(a) Runtime

(b) Output Size

Figure 2.12: Normalization $\mathcal{N}_{\{ssn\}}$ ($Incumbent$).

2.7.3 Normalization Attributes

In this experiment we evaluate the performance of temporal normalization with different normalization attributes. Splitting data across all start and end points independent of the normalization attributes not only violates change preservation for group based operators, but dramatically decreases the performance. We show this on the $Incumbent$ dataset and the following three normalization operations: $\mathcal{N}_{\{\}}$, $\mathcal{N}_{\{pcn\}}$ and $\mathcal{N}_{\{ssn\}}$. The runtime results and the output cardinality of these operations are shown in Fig. 2.13. There is a strong correlation between the normalization attributes and the performance. $\mathcal{N}_{\{\}}$ requires that all overlapping tuples are split, whereas $\mathcal{N}_{\{pcn\}}$ and $\mathcal{N}_{\{ssn\}}$ only require a split when the tuples match on the corresponding attribute values.



(a) Runtime (b) Output Size

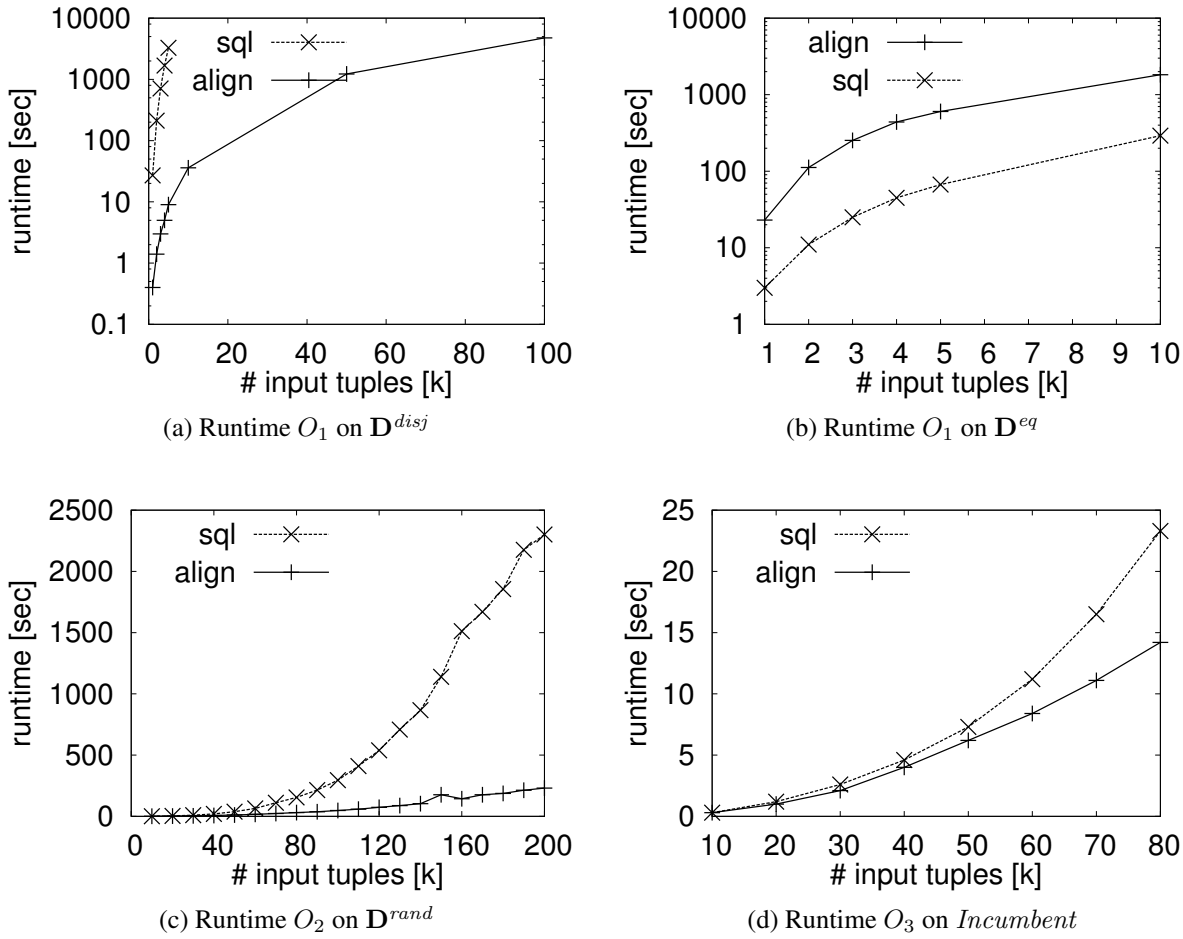
Figure 2.13: Normalizations ($Incumbent$).

2.7.4 Expressing Temporal Outer Joins in SQL

We compare the computation of temporal outer joins using temporal alignment ($align$) with the computation of temporal outer joins expressed in standard SQL (sql). To express a temporal outer join in SQL we have to express the join part using overlap predicates on timestamps and evaluate the negative part of the temporal outer join using joins and NOT EXISTS statements [Sno00]. The final result is the union of the two parts.

For the comparison we use three queries: $O_1 = \mathbf{r} \overset{T}{\ltimes}_{true} \mathbf{s}$, $O_2 = \mathbf{r} \overset{T}{\ltimes}_{Min \leq DUR(\mathbf{r}.T) \leq Max} \mathbf{s}$, and $O_3 = \mathbf{r} \overset{T}{\ltimes}_{\mathbf{r}.pcn=\mathbf{s}.pcn} \mathbf{s}$, and three synthetic datasets: $\mathbf{D}^{disj}$ where the intervals in both relations are disjoint, $\mathbf{D}^{eq}$ where the intervals in both relations are equal, and $\mathbf{D}^{rand}$ where we have random intervals and categories.

Figure 2.14a shows the runtime of query O_1 on $\mathbf{D}^{disj}$. As expected, *align* performs much faster than sql because of the NOT EXISTS used by SQL. The NOT EXISTS predicate is only efficient if a match can be found as fast as possible, so that the evaluation can terminate and return $false$. Since there are only few overlapping intervals in both relations, the NOT EXISTS has to scan almost the entire relation, which yields a quadratic complexity. The best setting for SQL for this is shown in Fig. 2.14b with the same query O_1 on dataset $\mathbf{D}^{eq}$. All timestamps of $\mathbf{D}^{eq}$ are equal, and thus the NOT EXISTS can be evaluated efficiently. For the $\mathbf{D}^{eq}$ dataset sql is more efficient than $align$ as it does not require any adjustment and the overhead is less than for alignment.



(a) Runtime O_1 on $\mathbf{D}^{disj}$

(b) Runtime O_1 on $\mathbf{D}^{eq}$

(c) Runtime O_2 on $\mathbf{D}^{rand}$

(d) Runtime O_3 on $Incumbent$

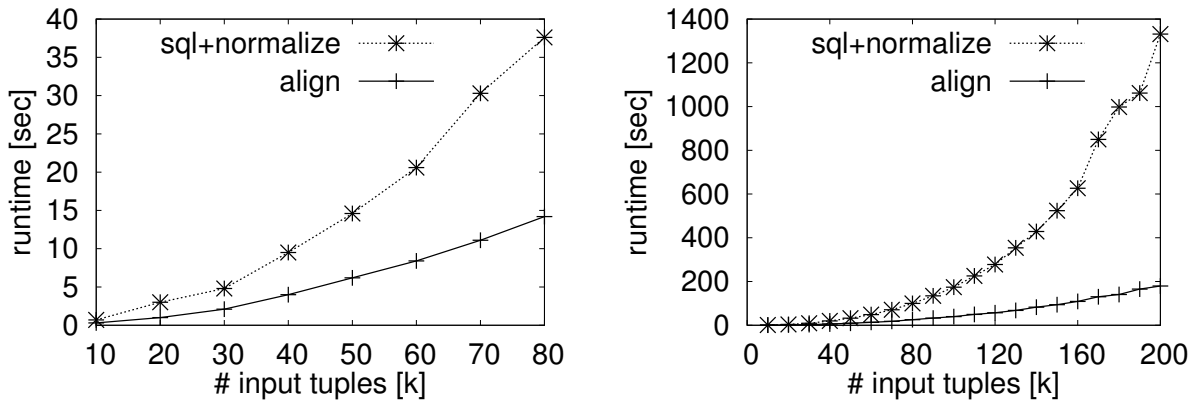
Figure 2.14: Outer Joins (Real World and Synthetic Data Sets).

Figure 2.14c shows the runtime of query O_2 on dataset $\mathbf{D}^{rand}$. The θ condition of the outer join does not allow efficient NOT EXISTS statements using antijoins, resulting in a high runtime of the SQL approach. The approach using temporal alignment performs much faster as it is more efficient for timestamp adjustment.

Finally, we run query O_3 on the real world $Incumbent$ dataset (Figure 2.14d). Both approaches are much faster than for the other datasets since the equality condition in the case of temporal alignment allows the database system to choose a fast nontemporal hash or merge join, and in the case of SQL to speed up the NOT EXISTS statements.

2.7.5 Expressing Temporal Outer Joins with SQL and Normalize

We compare the computation of temporal outer joins using temporal alignment (*align*) with an approach that expresses temporal outer joins using standard SQL plus temporal normalization for the negative part (*sql+normalize*). The joined part of the temporal outer join is computed with SQL, and temporal normalization is used for the temporal difference. Expressing outer joins with difference requires to compute the difference between an argument relation and the intermediate join result to determine all tuples that are not joining. Figure 2.15a shows the runtime behavior of query O_3 on the real world dataset $Incumbent$. *align* performs much faster than *sql+normalize* due to the expensive normalization steps that *sql+normalize* is required to perform on the intermediate join result. In the last experiment in Figure 2.15b we compare the



(a) Runtime O_3 on $Incumbent$

(b) Runtime O_3 on Random Dataset

Figure 2.15: Outer Joins (Real World and Synthetic Data Sets).

runtime of the same query O_3 on a random dataset. The interval timestamps have on average the same duration as in the $Incumbent$ dataset, but start and end timestamps are randomly distributed. This yields a larger temporal join result and more distinct splitting points than for the real world dataset. With a larger temporal join result and more candidate splitting points, the performance of *sql+normalize* decreases compared to *align*.

2.8 Conclusion and Future Work

In this paper we describe a relational algebra solution that provides native support for processing interval timestamped data with the sequenced semantics. We support the three properties

of the sequenced semantics (snapshot reducibility, extended snapshot reducibility and change preservation) through timestamp propagation and two temporal primitives, a temporal splitter and temporal aligner. With these primitives query processing becomes a two-step process: (1) propagate and adjust the interval timestamps of argument tuples such that changes are preserved and predicates and functions over the original timestamps remain possible, and (2) apply the corresponding nontemporal operator on the adjusted relations. We defined rules to reduce the operators of a temporal algebra to their nontemporal counterparts. We have implemented the temporal primitives and reduction rules in the kernel of PostgreSQL to get native database support for all operations of a temporal algebra, including outer joins, antijoins, and aggregations with predicates and functions over the original timestamps.

Future work includes the following directions: investigate indexing or merge sort techniques to improve the performance of the temporal primitives for cases when conventional join techniques cannot be evaluated efficiently; customize the temporal primitives for specific temporal operators to not produce adjusted tuples that do not contribute to the result for that operator (the current temporal primitives are generic and work for tuple and group based operators, respectively); extend the temporal primitives for a bag based temporal algebra.

CHAPTER 3

Query Time Scaling of Attribute Values

3.1 Introduction

In valid-time databases with interval timestamping each tuple is associated with a time interval over which the recorded fact is true in the modeled reality. The adjustment of these intervals is an essential part of processing interval timestamped data. Some attribute values remain valid if the associated interval changes, whereas others have to be scaled along with the time interval. For example, attributes that record total (cumulative) quantities over time, such as project budgets, total sales or total costs, often must be scaled if the timestamp is adjusted. *The goal of this demo is to show how to support the scaling of attribute values in SQL at query time.*

Whether an attribute value shall be scaled or not depends on the semantics of the query and is not a property of the schema. A query can ask for the *original value* or the *scaled value*, e.g., the sum of the budgets of all running projects (original values) or the sum of the available budgets for a specific time interval (scaled value). Therefore, a general solution must offer comprehensive support for scaling attribute values at query time. Applications must have the option to specify *whether* attribute values shall be scaled and *how* they should be scaled. This demo paper de-

scribes a solution that has been integrated into PostgreSQL and supports the scaling of attribute values at query time for all temporal operations.

Example 14. Consider project relation $\mathbf{proj}$ in Fig. 3.1. D is the department, N the project number, B the project budget, and $[T_S, T_E)$ the duration of the project. For instance, tuple r_1 records that project 1 in the DB department has a budget of 181K and runs from February 1, 2013 to July 31, 2013.

$\mathbf{proj}$

	D	N	B	T_S	T_E
r_1	DB	1	181K	2013/2/1	2013/8/1
r_2	DB	2	196K	2013/5/1	2014/1/1
r_3	AI	1	153K	2013/4/1	2013/9/1
r_4	AI	2	120K	2013/4/1	2013/9/1

$\mathbf{r_1}$

D	B	T_S	T_E
DB	89K	2013/2/1	2013/5/1
DB	165.6K	2013/5/1	2013/8/1
DB	122.4K	2013/8/1	2014/1/1
AI	273K	2013/4/1	2013/9/1

Figure 3.1: Query with a Scaling Function: $\mathbf{r_1} = {}_D\vartheta^T_{SUM(scale(B))}(\mathbf{proj})$.

To determine the available (time-varying) budget per department, we aggregate the budget for each department, i.e., $\mathbf{r_1} = {}_D\vartheta^T_{SUM(scale(B))}(\mathbf{proj})$. Here, ^{T} indicates a sequenced aggregation [DBG12] (i.e., the aggregation is performed at each point in time). $scale(B)$ indicates that the value of attribute B shall be scaled before the aggregation function is applied. The result of $\mathbf{r_1}$ is shown in Fig. 3.1. To get this result the original budget values must be scaled to the intervals of the result tuples and then aggregated. Note that the sum of B over all tuples (total budget) is the same in $\mathbf{proj}$ and $\mathbf{r_1}$ as required, i.e., $181 + 196 + 153 + 120 = 89 + 165.6 + 122.4 + 273$.

Our contributions are the following:

- We provide precise procedures to scale attribute values. The demo illustrates the mechanics of these procedures and allows to run temporal SQL queries with and without scaling.

- The support for scaling is comprehensive and not limited to a simple pre- or post-processing of values. Scaling is possible in grouping and join predicates, and in aggregate functions (cf. Example 14).

- We parametrize our solution with flexible user-defined functions that have been predefined and can be modified. The demo comes with simple scaling functions and more advanced scaling functions for trends.

- The scaling has been implemented in PostgreSQL 9.0, which is used for the demonstration and is available as open source software at http://www.ifi.uzh.ch/dbtg/research/align.html.

Related work: The scaling of attribute values in response to the adjustment of interval timestamps received scant attention and was considered a schema property. No implementations to systematically scale attribute values in response to changes of the associated interval timestamps have been provided. In Böhlen et al. [BGJ06b], three different attribute characteristics are proposed: *constant* attributes never change value during query processing, *malleable* attributes require an adjustment of the value when the timestamp changes, and *atomic* attributes become undefined (invalid) when the timestamp changes. For malleable attributes an adjustment function is proposed. We use the terminology from this work, propose an SQL implementation, and extend the work to scale attributes values in aggregate functions, grouping and join conditions. Böhlen et al. [BGJ06a] provide build-in support for malleable attributes for temporal aggregation. Support for malleable attributes in the grouping or for other temporal operators is not provided. Terenziani et al. [TS04] distinguishes between atelic facts that are valid for each point in time and telic facts that are only valid for one specific interval. The work focuses on the semantics of facts recorded in a database and proposes a three-sorted relational model (atelic, telic, nontemporal). In contrast, we provide a solution that allows the user to flexibly scale attribute values at query time, and we integrate support for scaling into a query language that adjusts intervals and allows to propagate the original intervals.

The rest of this paper is organized as follows. Section 3.2 gives the temporal relation algebra. Section 3.3 introduces scaling functions, and Section 3.4 describes the scaling procedure in temporal operations. Section 3.5 describes the demonstration scenarios.

3.2 Algebraic Basis

We assume a linearly ordered, discrete time domain, Ω^T. A time interval is a contiguous set of time points and is represented as a pair $T = [T_S, T_E) \in \Omega^T \times \Omega^T$, where T_S is the inclusive start point and T_E the exclusive end point. We associate each tuple with a time interval that represents the tuple's valid time. A temporal relation, $\mathbf{r}$, over schema $R = (A_1, \ldots, A_m, T)$ is a finite set of tuples over R. For a tuple r and an attribute A_i, $r.A_i$ denotes the value of A_i in r.

As abbreviations we write $\mathbf{A} = \{A_1, \ldots, A_m\}$ for all attributes of a relation, and $\mathbf{r}.\mathbf{A} = \mathbf{s}.\mathbf{A}$ for $\mathbf{r}.A_1 = \mathbf{s}.A_1 \wedge \ldots \wedge \mathbf{r}.A_m = \mathbf{s}.A_m$. We write $\mathbf{r}/\mathbf{s}$ to rename relation or attribute X to U.

We use two queries as running examples. The temporal aggregation query $\mathbf{r_1} = {}_D\vartheta^T_{SUM(scale(B))}(\mathbf{proj})$ is shown in Fig. 3.1. It computes the sum of attribute B over all tuples with equal D attribute over their common time interval. The temporal left outer join $\mathbf{r_2} = \mathbf{proj} \mathbin{⟕}^{T:scale(B)}_{D=R} \mathbf{q}$, which computes the left outer join between project relation $\mathbf{proj}$ and manager relation $\mathbf{q}$, is shown in Fig. 3.2. Here $^{T:scale(B)}$ indicates that attribute B shall be scaled in the result. It computes the left outer join between each tuple of $\mathbf{proj}$ and all tuples in $\mathbf{q}$ that satisfy the θ-condition $D = R$ and have a common time interval. A tuple's sub-interval in $\mathbf{proj}$ that does not have a match is reported with ω (NULL) values in the result.

proj

	D	N	B	T_S	T_E
r_1	DB	1	181K	2013/2/1	2013/8/1
r_2	DB	2	196K	2013/5/1	2014/1/1
r_3	AI	1	153K	2013/4/1	2013/9/1
r_4	AI	2	120K	2013/4/1	2013/9/1

q

R	M	T_S	T_E
DB	Ann	2013/5/1	2013/8/1
DB	Sam	2013/8/1	2014/1/1

$\mathbf{r_2}$

D	N	B	R	M	T_S	T_E
DB	1	89K	ω	ω	2013/2/1	2013/5/1
DB	1	92K	DB	Ann	2013/5/1	2013/8/1
DB	2	73.6K	DB	Ann	2013/5/1	2013/8/1
DB	2	122.4K	DB	Sam	2013/8/1	2014/1/1
AI	1	153K	ω	ω	2013/4/1	2013/9/1
AI	2	120K	ω	ω	2013/4/1	2013/9/1

Figure 3.2: Query with a Scaling Function: $\mathbf{r_2} = \mathbf{proj} \mathbin{⟕}^{T:scale(B)}_{D=R} \mathbf{q}$

A key element of our solution is a temporal algebra that separates the adjustment of interval timestamps from the actual operation [DBG12]. Two temporal primitives, *temporal normalization* $\mathcal{N}$ and *temporal alignment* Φ, are used to adjust the interval timestamps, followed by a call of the corresponding nontemporal operator on the adjusted tuples to obtain the final result. The reduction rules in Table 3.1 show the complete specification of the operators. For example, the temporal aggregation ${}_D\vartheta^T_{AVG(B)}(\mathbf{r})$ is reduced to ${}_{D,T}\vartheta_{AVG(B)}(\mathcal{N}_{\mathbf{r}.D=\mathbf{s}.D}(\mathbf{r}, \mathbf{r}/\mathbf{s}))$. For join operators a post-processing step (α) eliminates temporal duplicates over non-maximal time intervals [DBG12].

Operator			Reduction
Selection	$\sigma^T_\theta(\mathbf{r})$	=	$\sigma_\theta(\mathbf{r})$
Projection	$\pi^T_{\mathbf{B}}(\mathbf{r})$	=	$\pi_{\mathbf{B},T}(\mathcal{N}_{\mathbf{r.B}=\mathbf{s.B}}(\mathbf{r},\mathbf{r}/\mathbf{s}))$
Aggregation	${}_{\mathbf{B}}\vartheta^T_F(\mathbf{r})$	=	${}_{\mathbf{B},T}\vartheta_F(\mathcal{N}_{\mathbf{r.B}=\mathbf{s.B}}(\mathbf{r},\mathbf{r}/\mathbf{s}))$
Difference	$\mathbf{r} -^T \mathbf{s}$	=	$\mathcal{N}_{\mathbf{r.A}=\mathbf{s.A}}(\mathbf{r},\mathbf{s}) - \mathcal{N}_{\mathbf{r.A}=\mathbf{s.A}}(\mathbf{s},\mathbf{r})$
Union	$\mathbf{r} \cup^T \mathbf{s}$	=	$\mathcal{N}_{\mathbf{r.A}=\mathbf{s.A}}(\mathbf{r},\mathbf{s}) \cup \mathcal{N}_{\mathbf{r.A}=\mathbf{s.A}}(\mathbf{s},\mathbf{r})$
Intersection	$\mathbf{r} \cap^T \mathbf{s}$	=	$\mathcal{N}_{\mathbf{r.A}=\mathbf{s.A}}(\mathbf{r},\mathbf{s}) \cap \mathcal{N}_{\mathbf{r.A}=\mathbf{s.A}}(\mathbf{s},\mathbf{r})$
Cart. Prod.	$\mathbf{r} \times^T \mathbf{s}$	=	$\alpha((\phi_\top(\mathbf{r},\mathbf{s})) \bowtie_{\mathbf{r}.T=\mathbf{s}.T} (\phi_\top(\mathbf{s},\mathbf{r})))$
Inner Join	$\mathbf{r} \bowtie^T_\theta \mathbf{s}$	=	$\alpha((\phi_\theta(\mathbf{r},\mathbf{s})) \bowtie_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\phi_\theta(\mathbf{s},\mathbf{r})))$
Left O. Join	$\mathbf{r} ⟕^T_\theta \mathbf{s}$	=	$\alpha((\phi_\theta(\mathbf{r},\mathbf{s})) ⟕_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\phi_\theta(\mathbf{s},\mathbf{r})))$
Right O. Join	$\mathbf{r} ⟖^T_\theta \mathbf{s}$	=	$\alpha((\phi_\theta(\mathbf{r},\mathbf{s})) ⟖_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\phi_\theta(\mathbf{s},\mathbf{r})))$
Full O. Join	$\mathbf{r} ⟗^T_\theta \mathbf{s}$	=	$\alpha((\phi_\theta(\mathbf{r},\mathbf{s})) ⟗_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\phi_\theta(\mathbf{s},\mathbf{r})))$
Anti Join	$\mathbf{r} \triangleright^T_\theta \mathbf{s}$	=	$(\phi_\theta(\mathbf{r},\mathbf{s})) \triangleright_{\theta\wedge\mathbf{r}.T=\mathbf{s}.T} (\phi_\theta(\mathbf{s},\mathbf{r}))$

Table 3.1: Reduction Rules with Explicit Equality Predicates.

To provide access to the original interval timestamps of a tuple, we use *timestamp propagation* ϵ to propagate a copy of the timestamp as an explicit (nontemporal) attribute. Since the interval timestamps are adjusted, references to the original timestamp must be replaced by a reference to the propagated attribute. For example, to compute the average duration of projects, $\vartheta^T_{AVG(DUR(T))}(\mathbf{proj})$, the timestamp is replaced by the propagated timestamp, i.e., $\vartheta^T_{AVG(DUR(U))}(\epsilon_U(\mathbf{proj}))$ before applying the reduction rules. Timestamp propagation is used whenever the original timestamps are needed in θ conditions or aggregation functions, and for scaling where we need the original and the adjusted timestamps. The temporal primitives have been implemented in the kernel of PostgreSQL and are accessible as follows:

$$\epsilon_U(\mathbf{r}) : \texttt{SELECT Ts Us, Te Ue, * FROM r}$$
$$\mathcal{N}_\theta(\mathbf{r},\mathbf{s}) : \texttt{FROM (r NORMALIZE s ON } \theta \texttt{) r}$$
$$\phi_\theta(\mathbf{r},\mathbf{s}) : \texttt{FROM (r ALIGN s ON } \theta \texttt{) r}$$
$$\alpha(\mathbf{r}) : \texttt{SELECT ABSORB * FROM r}$$

In the rest of this paper we describe the integration of scaling for attribute values into SQL extended with these primitives.

3.3 Scaling Functions

When changing the timestamp that is associated with an attribute value x from an interval T_{old} to an interval T_{new} a scaling function can be used to adjust the value of x to the new timestamp. In this section, we define various scaling functions, that, given an attribute value x and two interval timestamps T_{new} and T_{old}, determine the scaled quantity x'.

3.3.1 Uniform Scaling

A function that scales attribute values proportionally to the length of the new timestamp is

$$scaleU(x, T_{new}, T_{old}) = x \cdot \frac{\mid T_{new} \mid}{\mid T_{old} \mid},$$

where $\mid T \mid$ denotes the duration of interval timestamp T.

The implementation of this scaling function with days as granularity in a user-defined function in PL/pgSQL is given below. The input parameters are an attribute value `x` of type `FLOAT` and four `DATE` parameters, of which the former two represent the new interval timestamp and the latter two the old interval timestamp.

```
CREATE OR REPLACE FUNCTION
scaleU(x FLOAT, ts_new DATE, te_new DATE,
                ts_old DATE, te_old DATE)
RETURNS FLOAT AS $$
BEGIN
 RETURN x * (te_new - ts_new) / (te_old - ts_old);
END; $$
LANGUAGE PLPGSQL;
```

3.3.2 Atomic Scaling

Attributes with atomic characteristics [BGJ06b] are attribute values that shall not be scaled to different intervals and thus become invalid when the associated timestamp is adjusted. This can

be handled by a scaling function that returns the value ω (NULL) whenever the new and old timestamps are different:

```
CREATE OR REPLACE FUNCTION
scaleA(x FLOAT, ts_new DATE, te_new DATE,
                ts_old DATE, te_old DATE)
RETURNS FLOAT AS $$
BEGIN
 IF ts_new = ts_old AND te_new = te_old THEN
  RETURN x;
 END IF;
 RETURN NULL;
END; $$
LANGUAGE PLPGSQL;
```

3.3.3 Trend Scaling

Attribute values that shall be scaled are not always uniformly distributed over the interval timestamp. Sometimes the distribution of attribute values follows a *trend*, represented by a function $f(t)$. Given an attribute value x and two time intervals T_{old} and T_{new}, we can define a scaling function over the integrals of the trend function:

$$scaleT(x, T_{new}, T_{old}) = x \cdot \frac{\int_{TS_{new}}^{TE_{new}} f(t) \cdot dt}{\int_{TS_{old}}^{TE_{old}} f(t) \cdot dt}.$$

Assume we want to scale according to the costs of power consumption, which fluctuates by 20% due to cooling. The temperature over a year follows a cosine trend and peaks during summer. We get the following trend function: $f(t) = 1 + \frac{\cos(2*\pi*off/365)}{10}$, where off is the offset between t and the peak, i.e., $off = t -$ '2013/7/15', shown in Fig. 3.3.

The integral for this trend function is:

$$\int_{off_s}^{off_e} f(t) \cdot dt = (off_e - off_s) + 365/(20 \cdot \pi) \cdot (\sin(2 \cdot \pi \cdot off_e/365) - \sin(2 \cdot \pi \cdot off_s/365))$$

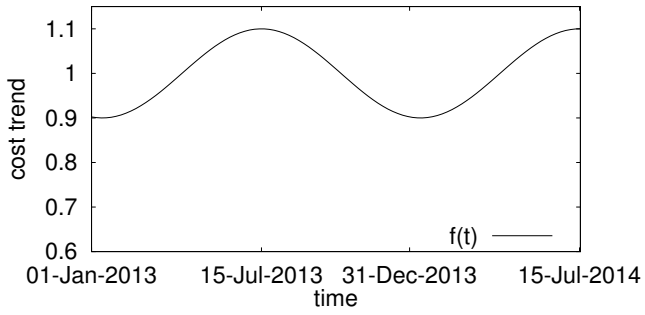


Figure 3.3: Trend function $f(t)$.

and we implement the following scaling function in PL/pgSQL, that scales an attribute value according to the ratio of the new and old integral (weight):

```
CREATE OR REPLACE FUNCTION
scaleT(x FLOAT, ts_new DATE, te_new DATE,
                ts_old DATE, te_old DATE)
RETURNS FLOAT AS $$
DECLARE
 w_old FLOAT; w_new FLOAT;
 off_s INT; off_e INT;
BEGIN
 off_s := ts_old-'2013/7/15';
 off_e := te_old-'2013/7/15';
 w_old := (off_e - off_s) + 365 / (20 * pi()) *
  (sin((2*pi()*off_e) / 365) - sin((2*pi()*off_s) / 365));
 off_s = ts_new-'2013/7/15';
 off_e = te_new-'2013/7/15';
 w_new := (off_e - off_s) + 365 / (20 * pi()) *
  (sin((2*pi()*off_e) / 365) - sin((2*pi()*off_s) / 365));
 RETURN x * w_new / w_old;
END; $$
LANGUAGE PLPGSQL;
```

To make sure that scaling functions are deterministic, functions must be strictly positive.

proj

	D	N	B	T_S	T_E
r_1	DB	1	181K	2013/2/1	2013/8/1
r_2	DB	2	196K	2013/5/1	2014/1/1
r_3	AI	1	153K	2013/4/1	2013/9/1
r_4	AI	2	120K	2013/4/1	2013/9/1

(a) Relation **proj**.

r1

D	B	T_S	T_E
DB	89K	2013/2/1	2013/5/1
DB	165.6K	2013/5/1	2013/8/1
DB	122.4K	2013/8/1	2014/1/1
AI	273K	2013/4/1	2013/9/1

(b) Uniform Scaling (`scaleU`).

r1

D	B	T_S	T_E
DB	ω	2013/2/1	2013/5/1
DB	ω	2013/5/1	2013/8/1
DB	ω	2013/8/1	2014/1/1
AI	273K	2013/4/1	2013/9/1

(c) Atomic Scaling (`scaleA`).

r1

D	B	T_S	T_E
DB	83.6K	2013/2/1	2013/5/1
DB	174.7K	2013/5/1	2013/8/1
DB	118.7K	2013/8/1	2014/1/1
AI	273K	2013/4/1	2013/9/1

(d) Trend Scaling (`scaleT`).

Figure 3.4: Query $\mathbf{r_1} = {}_D\vartheta^T_{SUM(scale(B))}(\mathbf{proj})$ with Different Scaling Functions.

3.4 Scaling Attribute Values in Operations

This section defines how to scale attribute values in SQL. The basic approach is to substitute the occurrence of attributes that shall be scaled through a scaling function. The procedures are different for normalization and alignment since operations following a normalization do not allow propagated timestamps and for alignment duplicate elimination (α) must be done before scaling [DBG12]. Note that the scaling of values at the end is not sufficient since scaled values might be used, e.g., in grouping, aggregation functions and join predicates.

3.4.1 Projection, Aggregation and Set Operations

For $\psi^T \in \{\pi^T, \vartheta^T, -^T, \cup^T, \cap^T\}$ the *temporal normalization* operator $\mathcal{N}$ is used to adjust the timestamps (cf. Table 3.1) and scaling can be used as follows:

1. propagate timestamps, ϵ;
2. normalize, $\mathcal{N}_\theta$ (possibly scale values);
3. possibly scale values and remove propagated attributes;

4. apply the corresponding nontemporal operator, ψ.

First, the timestamps are propagated using ϵ since the original timestamp is needed later for scaling. Second, the relations are adjusted using $\mathcal{N}_\theta$. If θ contains an attribute that needs to be scaled, expressions in θ of the form $\mathbf{r}.B = \mathbf{s}.B$ are substituted with $scale(\mathbf{r}.B, T, U) = scale(\mathbf{s}.B, T, V)$, where U and V are the propagated timestamps of $\mathbf{r}$ and $\mathbf{s}$, respectively, and $T = U \cap V$. Third, attributes that need to be scaled in the result of step 2 are scaled according to the adjusted and original timestamps by substituting expressions B with $scale(B, T, U)$; propagated timestamps are now removed. Finally, the corresponding nontemporal operator $\psi \in \{\pi, \vartheta, -, \cup, \cap\}$ is applied.

3.4.2 Cartesian Product, Inner, Outer and Anti Joins

For $\psi^T \in \{\times^T, \bowtie^T, ⟕^T, ⟖^T, ⟗^T, \triangleright^T\}$ the *temporal alignment* operator is used to adjust the timestamps (ref. Table 3.1) and scaling can be used as follows:

1. propagate timestamps, ϵ;
2. align, ϕ_θ (possibly scale values);
3. apply nontemporal operator, ψ (possibly scale values);
4. possibly scale values and remove propagated attributes.

In step 3 the reduction according to Table 3.1 is done before scaling and removing propagated timestamps. The last step is to scale attribute values and to remove propagated timestamps (unless they are used in subsequent operations).

3.5 Demonstration Scenario

In the demonstration we show step-by-step how to scale attribute values in temporal operations, using our implementation in PostgreSQL. We show a number of different scenarios, including the ones described below. Additionally, we provide a set of sample relations and encourage people to construct alternative query expressions that can be evaluated.

3.5.1 Scenario 1

Consider query $\mathbf{r_1} = {}_D\vartheta^T_{SUM(scale(B))}(\mathbf{proj})$ of Example 14. The individual steps of the scaling procedure are as follows (cf. Sec. 3.4.1):

1. $\mathbf{p_1} \leftarrow \epsilon_U(\mathbf{proj})$
2. $\mathbf{p_2} \leftarrow \mathcal{N}_{\mathbf{x}.D=\mathbf{y}.D}(\mathbf{p_1}/\mathbf{x}, \mathbf{p_1}/\mathbf{y})$
3. $\mathbf{p_3} \leftarrow \pi_{D,N,scale(B,T,U)/B,T}(\mathbf{p_2})$
4. $\mathbf{r_1} \leftarrow {}_{D,T}\vartheta_{SUM(B)}(\mathbf{p_3})$

First, the timestamps of relation $\mathbf{proj}$ are propagated. Second, the normalization primitive is applied; no scaling in the θ-condition is required since D shall not be scaled. Third, a generalized projection is used to scale the attribute value of B and to remove the propagated timestamp U. Finally, the nontemporal aggregation operator is applied to the intermediate result of step 3. The mapping of this algebra expression to SQL is as follows:

```
WITH
p1 AS (SELECT Ts Us, Te Ue, * FROM p),
p2 AS (SELECT * FROM (p1 x NORMALIZE p1 y ON x.D=y.D) p),
p3 AS (SELECT D, N, scaleU(B, Ts, Te, Us, Ue) B, Ts, Te
       FROM p2)

SELECT   D, SUM(B), Ts, Te
FROM     p3
GROUP BY D, Ts, Te;
```

Fig. 3.4 shows the result of Query $\mathbf{r_1}$, with three different scaling functions on the same input relation $\mathbf{proj}$. In all three cases, the first result tuple was derived from r_1 adjusted to period $[2013/2/1, 2013/5/1)$. With uniform scaling (3.3.1) we get $B = 89\text{K}$. With atomic scaling (3.3.2), we get $B = \omega$ since the interval timestamp of the result tuple changes. In the case of trend scaling (3.3.3), we get $B = 83.6\text{K}$, which is less than with uniform scaling since the weight during the winter period is lower than during summer.

3.5.2 Scenario 2

We determine the budgets of projects for the responsible manager. To include project budgets without manager, a temporal left outer join between project relation $\mathbf{proj}$ and manager relation $\mathbf{q}$ on the department is required. The step by step procedure for query $\mathbf{r_2} = \mathbf{proj} ⟕^{T:scale(B)}_{D=R} \mathbf{q}$ is as follows (cf. Sec. 3.4.2):

1. $\mathbf{p_1} \leftarrow \epsilon_U(\mathbf{proj})$
   $\mathbf{q_1} \leftarrow \epsilon_V(\mathbf{q})$

2. $\mathbf{p_2} \leftarrow \phi_{D=R}(\mathbf{p_1}, \mathbf{q_1})$
   $\mathbf{q_2} \leftarrow \phi_{D=R}(\mathbf{q_1}, \mathbf{p_1})$

3. $\mathbf{pq} \leftarrow \alpha(\mathbf{p_2} ⟕_{D=R \wedge \mathbf{p_2}.T=\mathbf{q_2}.T} \mathbf{q_2})$

4. $\mathbf{r_2} \leftarrow \pi_{D,N,scale(B,T,U)/B,R,M,T}(\mathbf{pq})$

After timestamp propagation, both relations are aligned on the departments (i.e., the join condition). Next, the nontemporal left-outer join is applied, followed by the scaling of attribute value B. The mapping to SQL is as follows:

```
WITH
p1 AS (SELECT Ts Us, Te Ue, * FROM p),
q1 AS (SELECT Ts Vs, Te Ve, * FROM q),
p2 AS (SELECT * FROM (p1 ALIGN q1 ON D=R) p2),
q2 AS (SELECT * FROM (q1 ALIGN p1 ON D=R) q2),
pq AS (SELECT ABSORB D, N, B, Us, Ue, M, R,
              Vs, Ve, p2.Ts, p2.Te
       FROM p2 LEFT OUTER JOIN q2
       ON D=R AND p2.Ts=q2.Ts AND p2.Te=q2.Te)

SELECT D, N, scaleU(B, Ts, Te, Us, Ue) B, R, M, Ts, Te
FROM pq;
```

Query $\mathbf{r_2}$ with uniform scaling is illustrated in Fig. 3.2. The first result tuple is derived from project 1 of the DB department. The time interval is $[2013/2/1, 2013/5/1)$ the scaled budget is 89K, and no manager was responsible. The second result tuple is derived from project 1 of the DB department and manager Ann over their common time interval $[2013/5/1, 2013/8/1)$ with a scaled budget of 92K.

Acknowledgements

This work was supported in part by the Swiss National Science Foundation (SNSF) through the Tameus project (proposal no 200021 135361).

CHAPTER 4

Overlap Interval Partition Join

Abstract

Each tuple in a valid-time relation includes an interval attribute T that represents the tuple's valid time. The overlap join between two valid-time relations determines all pairs of tuples with overlapping intervals. Although overlap joins are common, existing partitioning and indexing schemes are inefficient if the data includes long-lived tuples or if intervals intersect partition boundaries.

We propose *Overlap Interval Partitioning* ($\mathcal{OIP}$), a new partitioning approach for data with an interval. $\mathcal{OIP}$ divides the time range of a relation into k base granules and defines overlapping partitions for sequences of contiguous granules. $\mathcal{OIP}$ is the first partitioning method for interval data that gives a *constant clustering guarantee*: the difference in duration between the interval of a tuple and the interval of its partition is independent of the duration of the tuple's interval. We offer a detailed analysis of the *average false hit ratio* and the *average number of partition accesses* for queries with overlap predicates, and we prove that the average false hit ratio is independent of the number of short- and long-lived tuples. To compute the overlap join, we propose the *Overlap Interval Partition Join* (OIPJOIN), which uses $\mathcal{OIP}$ to partition the input

relations on-the-fly. Only the tuples from overlapping partitions have to be joined to compute the result. We analytically derive the optimal number of granules, k, for partitioning the two input relations, from the size of the data, the cost of CPU operations, and the cost of main memory or disk IOs. Our experiments confirm the analytical results and show that the OIPJOIN outperforms state-of-the-art techniques for the overlap join.

4.1 Introduction

A key operation in valid-time databases is the *overlap join* [GJSS05]: given two valid-time relations $\mathbf{r}$ and $\mathbf{s}$, find all pairs of tuples $r \in \mathbf{r}$ and $s \in \mathbf{s}$ with overlapping intervals, i.e., $r.T \cap s.T$. The overlap join gives the query optimizer an efficient option if other predicates are absent, exhibit a poor selectivity, or must be evaluated after the overlapping interval has been computed. For instance, to find employees who are employed during at least 5 months when a project is ongoing, we first must determine the overlapping interval between an employee and a project, and then check that the duration of the overlapping interval is at least 5 months. Our goal is an efficient join for interval data that offers the query optimizer a viable option when other joins do not perform well.

Partitioning techniques for interval data associate each partition with a *partition interval*. Each tuple is stored in the best fitting partition, i.e., the partition interval must cover the interval of the tuple, and there may not exist a smaller partition interval that covers the interval of the tuple. As an example, consider a partition p with partition interval $[2012\text{-}1, 2012\text{-}4]$ and a tuple s with interval $[2012\text{-}2, 2012\text{-}3]$. Tuple s can be stored in partition p since 2012-2 $\geq$ 2012-1 and 2012-3 $\leq$ 2012-4, and it is indeed stored in p if and only if there is no other partition with a smaller partition interval that covers the interval of s. Since a partition interval is usually larger than the intervals of the tuples in this partition, we get *false hits* when searching in a partition for tuples that overlap a query interval (a false hit is a tuple that is fetched with a partition but does not contribute to the result). False hits increase the number of IOs, since more data must be fetched, and the number of CPU operations, since false hits must be detected and discarded. In order to reduce the number of false hits, it is possible to create more partitions. Many partitions, however, increase the number of IOs since we get more partially filled blocks. This increases the number of CPU operations for *searching and navigating* in the access structure.

This paper proposes the OIPJOIN, together with *Overlap Interval Partitioning* ($\mathcal{OIP}$), to efficiently compute the overlap join. $\mathcal{OIP}$ partitions the time range of a relation uniformly at a granularity that is given by k temporally disjoint granules of duration d. We create partitions for all sequences of adjacent granules. This approach gives a *constant clustering guarantee*, i.e., the difference in duration between a tuple and its partition is less than $2d$, independent of the duration of the tuple. The access structure of $\mathcal{OIP}$, termed *lazy partition list*, omits empty partitions without sacrificing performance or functionality. The OIPJOIN is *self-adjusting*, i.e, it automatically determines the optimal number of granules, k, that minimizes the overhead costs of the OIPJOIN.

Example 15. Figure 4.1 illustrates $\mathcal{OIP}$ with $k = 4$ granules for two relations $\mathbf{r}$ and $\mathbf{s}$. The time range of $\mathbf{r}$ is $[2012\text{-}5, 2012\text{-}11]$, and the granules have a duration of $\lceil \frac{2012\text{-}11 - 2012\text{-}5 + 1}{4} \rceil = \lceil \frac{7}{4} \rceil = 2$ months. This is the granularity at which relation $\mathbf{r}$ is partitioned. The partitions span 2, 4, 6, or 8 months. Similarly, relation $\mathbf{s}$ is partitioned over its time range $[2012\text{-}1, 2012\text{-}12]$. The granules have a duration of 3 months, and the partitions span 3, 6, 9, or 12 months. For the OIPJOIN, $\mathbf{r} \bowtie_{\mathbf{r}.T \cap \mathbf{s}.T} \mathbf{s}$, we process for each partition in $\mathbf{r}$ the overlapping (relevant) partitions in $\mathbf{s}$. For instance, for the partition that contains r_1 and r_2, three partitions in $\mathbf{s}$ are processed, yielding three false hits, namely $\{s_6\}$ for r_1 and $\{s_3, s_5\}$ for r_2. For the partition that contains r_3, two partitions in $\mathbf{s}$ are processed, and there are no false hits. Overall, five partitions of $\mathbf{s}$ are accessed with three false hits and eight result tuples.

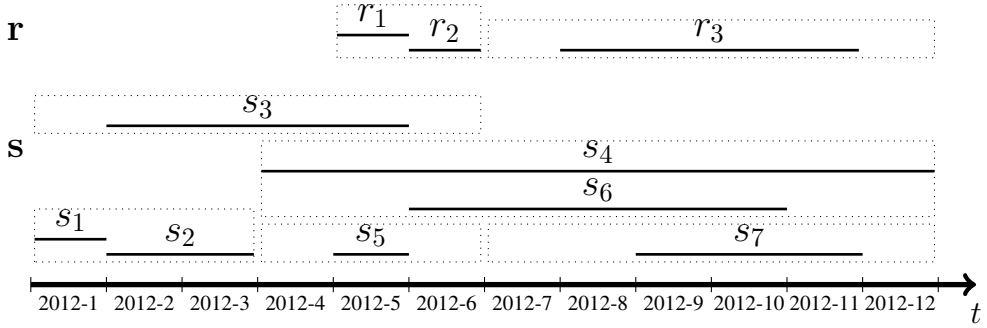


Figure 4.1: Sample Relations and $\mathcal{OIP}$.

False hits and partition accesses incur overhead costs for the OIPJOIN. The number of false hits and the number of partition accesses are inversely related. Increasing the number of granules k (i.e., shorter granules and more partitions) increases the number of partition accesses, but decreases the number of false hits, and vice versa. We analytically derive the k that minimizes the overhead costs by adapting to the size of the two relations, the cost of CPU operations, and the

cost of IOs. Instead of assuming a dominating cost factor, we propose a cost model that accounts for CPU and IO costs. Note that IO costs can be memory IOs or disk IOs. A main memory IO is faster than a disk IO, but slower than a CPU operation [SGG12]. Since data are transferred in chunks from the memory to the processor, it is favorable to store tuples in contiguous main memory blocks.

Summarizing, our technical contributions are as follows:

- We introduce $\mathcal{OIP}$ as partitioning strategy for the OIPJOIN. $\mathcal{OIP}$ offers a *constant clustering guarantee*, which ensures that the join does not deteriorate. The difference in duration between a tuple and its partition is less than two granules.
- We provide a detailed analysis of the *average false hit ratio* (AFR) and the *average number of partition accesses* (APA) for $\mathcal{OIP}$. We prove that AFR for uniformly distributed query intervals is smaller than $\frac{1}{k}$ and independent of the number of short- and long-lived tuples.
- The OIPJOIN is *self-adjusting*, i.e, it automatically determines the optimal number of granules k. We develop a cost function for the OIPJOIN and minimize this cost function to get the optimal number of granules k to partition the relations. k minimizes the overhead costs due to false hits and partition accesses for IO costs $c\_io \geq 0$ and CPU costs $c\_cpu \geq 0$.
- We describe an implementation of the OIPJOIN based on $\mathcal{OIP}$ and compare it with self-adjusting overlap joins based on quadtree, loose quadtree, segment tree, and relational interval tree. The experiments confirm that the OIPJOIN outperforms these approaches.

The rest of the paper is organized as follows. Section 4.2 discusses related work. Section 4.3 provides preliminaries. Section 4.4 describes overlap interval partitioning ($\mathcal{OIP}$) and its implementation. Section 4.5 analytically investigates the average false hit ratio (AFR) and the average number of partition accesses (APA) of $\mathcal{OIP}$. Section 4.6 describes the OIPJOIN and derives the optimal value for k. Section 4.7 reports the results of our empirical evaluation.

4.2 Related Work

We describe, in turn, related work on (a) self-adjusting approaches that, as the OIPJOIN, adapt to the data and do not require user-specified parameters; (b) parameter-guided approaches that

can/must be tuned with application-specific parameters; and (c) disk-based approaches that introduced some of the key concepts used in later works.

4.2.1 Self-Adjusting Approaches

The quadtree [FB74; Sam05] recursively subdivides the space into cells and places objects in the smallest enclosing cell.[1] Since split boundaries are propagated down the tree and objects are not allowed to overlap boundaries, small objects that overlap split boundaries end up high in the tree. Therefore the quadtree does not have a clustering guarantee. For instance, time range $[1, 32]$ is recursively split into $[1, 16]$ and $[17, 32]$ and so on, and a tuple with interval $[16, 17]$ is placed in the root. This yields many false hits for overlap predicates since all tree nodes that overlap the query interval need to be scanned. The quadtree relies on a hierarchical tree structure, and in order to navigate to nodes at lower levels, all parent nodes must be stored, even if they are empty. To avoid many partially filled blocks, density based splitting is used, which propagates tuples down the tree only when blocks are full. This, however, increases the number of false hits.

The loose quadtree [Sam05; SSA13] addresses the limitations of the quadtree for small objects. It permits at each level partially overlapping cells. The amount of overlapping is determined by a user-specified cell expansion factor, $p > 0$, where $p = 1$ is widely accepted as the best value [Ulr00; Sam05]. An expanded cell has width $(1 + p) \cdot w$, where w is the width of a quadtree cell. For instance, time range $[1, 32]$ is recursively split into $[1, 24]$ and $[9, 32]$ and so on. A tuple with interval $[16, 17]$ is placed in either $[14, 17]$ or $[16, 19]$, which are the expanded quadtree cells for $[15, 16]$ and $[17, 18]$. The join performance deteriorates for long-lived tuples since the time ranges grow with a factor of two, i.e., the number of partitions for long-lived tuples is much lower than for small tuples. The loose quadtree provides a clustering guarantee that is not constant. The guarantee depends on the duration of a tuple and is weaker for longer tuples. For instance, for $p = 1$ and a relation that spans 2000 days, a tuple of duration 80 days can be in a partition that spans 250 days, yielding a difference of 170 days between the tuple and the associated partition. A tuple that spans 282 days can even be in a partition of 1000 days, which is a difference of 718 days.

The relational interval tree [KPS00] implements Edelsbrunner's interval tree on top of a relational DBMS. The approach uses two B+-tree indices to index intervals according to a key and start

[1] In order to manage intervals with 2D access structures we omit the second dimension, which reduces 2D points and 2D rectangles to, respectively, 1D points and intervals.

point and end point. A query interval is first transformed into a key point list and a key range list, which in a second step are joined with the B+-tree indices. For instance, given an indexed time range of $[1, 64]$ and a query interval $[5, 7]$, the key point list is $\{32, 16, 8\}$ and the key range list $\{[4, 4], [5, 7]\}$. These lists are joined with the help of the two B+-tree indices to get the final result. Various join techniques based on the relational interval tree have been proposed [EHS04], such as an Index-Based Loop Join and several partition based joins (Up-Down, Down-Down, Up-Up depending on tree traversal). In all these join techniques, long-lived tuples lead to many CPU operations since a large number of nodes must be joined. To get a better IO performance for block storage, the data can be clustered according to an index on either the start or the end points. Nevertheless, long-lived tuples deteriorate the performance since for long-lived tuples the clustering of the two indices varies more than for short-lived tuples.

The segment tree [Ben77; BKOS00] is an indexing technique for intervals. It builds disjoint segments (intervals) at the leaf level, using all start and end points in a relation. Internal nodes merge all segments of their children nodes. A tuple is associated with all sub-tree roots whose segment is completely covered by the tuple's interval. The segment tree efficiently retrieves all tuples that include a given time point. In order to compute an overlap join, possibly empty parent nodes must be scanned, and duplicated tuples that are assigned to multiple nodes must be fetched (IO cost) and identified (CPU cost). This is particularly expensive for long-lived tuples. For instance, for a relation with three tuples r_1, r_2, and r_3 with intervals $[1, 5]$, $[3, 9]$, and $[8, 9]$, respectively, we have at the leaf level (level 2) the four segments $[1, 2]$, $[3, 5]$, $[6, 7]$, and $[8, 9]$. At level 1, we have the segments $[1, 5]$ and $[6, 9]$, and at level 0 (root) the segment $[1, 9]$. Tuple r_2 is stored twice, namely in $[3, 5]$ and $[6, 9]$, and it must be read twice for the query interval $[5, 6]$.

4.2.2 Parameter-Guided Approaches

In [LOT94], a spatially partitioned temporal join is proposed, where interval data is mapped to points in a two dimensional grid. Partitions are regions in the plane. Two relations are joined by determining for each partition of the outer relation the relevant partitions of the inner relation. Two implementations are proposed, namely to store partitions physically on disk blocks or to use spatial indices to index the regions of partitions. While existing spatial indices can be reused, long-lived tuples substantially increase the number of index nodes to scan. The number of partitions must be specified by the application.

The snapshot index [TK95] is an access method for disk resident transaction-time databases. Intervals in transaction-time databases are clustered since database modifications occur in increasing time order. Blocks are distinguished by usefulness according to an application parameter a that indicates the number of false hits a block is allowed to generate. Long-lived tuples are artificially deleted and re-inserted using controlled splits. Splitting is not a general solution for valid-time databases since it changes the meaning of tuples [DBG12; DBG13]. Parameter a must be chosen as a tradeoff between artificial splits and false hits.

In [NTH+13], an approach is proposed for data that is stored in main memory. It is similar to the spatial hash join [LR96] and uses an R-Tree to group tuples into minimum bounding rectangles (MBRs). The tuples of one relation are stored in the leaf nodes, and the tuples of the other relation in the lowest internal node, where more than one child's MBR overlaps the tuple. The join is performed by joining leaf with internal nodes. The approach aims to reduce the number of CPU comparisons and requires three parameters: tree fanout, number of partitions, and cells per dimension.

4.2.3 Disk-Based Approaches

The size separation spatial join [KS97] is a partitioning strategy for the overlap join of disk resident spatial data and is similar to the quadtree. Instead of using a tree structure, the levels of the tree are mapped to sorted files. The join is then performed by a synchronized scan of two sorted files that represent the partitioned relations. The approach reduces IO and provides a good filling of blocks, but, due to the recursive space division, small objects are not guaranteed to be stored at a low level. Thus, the size separation spatial join has no clustering guarantee and may produce many false hits.

The grace partition join [SSJ94] is used for valid-time natural joins of disk resident data, i.e., overlap joins with additional equality predicates. It samples the relations to determine the partitions for the tuple intervals. A tuple is stored in the last partition it overlaps. Partitions are joined from the last to the first partition. Long-lived tuples that overlap several partitions are migrated to the next partition during join processing. The approach is only efficient for few long-lived tuples, where the overhead of migration is low.

The R*-tree [BKSS90; BS09] uses MBRs to group nearby objects and stores object IDs in the leaf nodes. The internal nodes build an index on the leaf nodes using MBRs. MBRs of both

leaf and internal nodes might overlap. The tree is expensive to construct due to the propagation of MBRs. Long-lived tuples increase the MBRs and produce more false hits (page faults). For overlap joins, it is necessary to follow multiple paths in the R*-tree.

4.3 Preliminaries

We assume a discrete time domain, Ω^T, consisting of a linearly ordered set of time points. An interval T is a contiguous set of time points and is represented as a pair $[T_S, T_E]$, where T_S is the inclusive start point and T_E the inclusive end point. We use the following operations on time points and intervals: $x \in T$ if time point x is contained in interval T, i.e., $T_S \leq x \leq T_E$; $Q \cap T$ if Q and T intersect, i.e., there exists a time point x such that $x \in Q \wedge x \in T$; $T \subseteq U$ if interval T is contained in interval U, i.e., $\forall x \in T \Rightarrow x \in U$; $T_E - T_S$ determines the difference in number of time points between T_S and T_E; $T_S + x$ shifts time point T_S by x time points to the right, i.e., $T_E - T_S = x \Rightarrow T_S + x = T_E$; and $|T| = (T_E - T_S) + 1$ is the duration (length) of interval T.

We use tuple timestamping and associate each tuple with a single interval that represents the tuple's valid time. A temporal relation schema is represented as $R = (A_1, \ldots A_m, T)$, where $A_1 \ldots A_m$ are attributes with domain Ω_i and T is the interval attribute over $\Omega^T \times \Omega^T$. A tuple r over schema R is a finite set that contains for every A_i a value $v_i \in \Omega_i$ and for T an interval $[T_S, T_E] \in \Omega^T \times \Omega^T$. A temporal relation $\mathbf{r}$ over schema R is a finite set of tuples over R. A valid-time relation $\mathbf{r}$ spans time range $U = [U_S, U_E]$ if U_S is the smallest start time point of any tuple in $\mathbf{r}$ and U_E the largest end time point of any tuple in $\mathbf{r}$. We write l for the duration of the longest tuple in a relation $\mathbf{r}$, and λ for the duration of the longest tuple as a fraction of the time range, i.e., $\lambda = l/|U|$. We use indices r and s to distinguish between the outer and inner relation in joins, e.g., n_r and n_s are, respectively, the cardinality of the outer and inner relation in $\mathbf{r} \bowtie \mathbf{s}$.

4.4 Overlap Interval Partitioning

In this section, we first define Overlap Interval Partitioning ($\mathcal{OIP}$) and show how the relevant partitions, i.e., partitions that overlap a query interval, are calculated. Second, we establish the constant clustering guarantee. Third, we show how to manage physical partitions of $\mathcal{OIP}$ with a lazy partition list that omits unused partitions.

4.4.1 Definition

$\mathcal{OIP}$ divides a time range U into k equally sized granules of duration d, which define the base granularity of the partitioning (we discuss in Section 4.6.2 how to derive k).

Definition 13. (*$\mathcal{OIP}$ Configuration*) Let $\mathbf{r}$ be a temporal relation with time range $U = [U_S, U_E]$. An *$\mathcal{OIP}$ configuration* for a given k is a triple (k, d, off), where $d = \lceil \frac{|U|}{k} \rceil$ is the duration of each granule and $\mathit{off} = U_S$ is the start point of the partitioned time range.

A partition interval spans a sequence of one or more adjacent granules. A tuple is assigned to the smallest partition whose partition interval completely covers the tuple's interval.

Definition 14. (*$\mathcal{OIP}$ Partition*) Let $\mathbf{r}$ be a temporal relation with $\mathcal{OIP}$ configuration (k, d, off). An *$\mathcal{OIP}$ partition*, $p_{i,j}$, with $0 \leq i \leq j < k$, spans all granules from i to j and has the partition interval $p_{i,j}.T = [\mathit{off} + i \cdot d,\ \mathit{off} + (j+1) \cdot d - 1]$. A tuple $r \in \mathbf{r}$ is placed in partition $p_{i,j}$ iff $\lfloor \frac{r.T_S - \mathit{off}}{d} \rfloor = i$ and $\lfloor \frac{r.T_E - \mathit{off}}{d} \rfloor = j$.

Example 16. Relation $\mathbf{s}$ in Figure 4.2 includes seven tuples and spans the time range $U = [\text{2012-1}, \text{2012-12}]$. The $\mathcal{OIP}$ configuration with $k = 4$ granules is $(4, 3, \text{2012-1})$ with granule duration $d = \lceil \frac{|U|}{k} \rceil = \lceil \frac{12}{4} \rceil = 3$ months and start time point $\mathit{off} = U_S = \text{2012-1}$. The partitions that span one granule are $p_{0,0}$, $p_{1,1}$, $p_{2,2}$, and $p_{3,3}$, each ranging over three months. The partitions $p_{0,1}$, $p_{0,2}$, $p_{0,3}$, $p_{1,2}$, $p_{1,3}$, and $p_{2,3}$ span more than one granule each, e.g., partition $p_{0,1}$ spans the range $[\text{2012-1}, \text{2012-6}]$. Tuple s_1 is placed in partition $p_{0,0}$ since $\lfloor \frac{s_1.T_S - \mathit{off}}{d} \rfloor = \lfloor \frac{\text{2012-1} - \text{2012-1}}{3} \rfloor = \lfloor \frac{0}{3} \rfloor = 0$ and $\lfloor \frac{s_1.T_E - \mathit{off}}{d} \rfloor = \lfloor \frac{\text{2012-1} - \text{2012-1}}{3} \rfloor = \lfloor \frac{0}{3} \rfloor = 0$. Tuple s_6 is placed in partition $p_{1,3}$ since $\lfloor \frac{s_6.T_S - \mathit{off}}{d} \rfloor = \lfloor \frac{\text{2012-6} - \text{2012-1}}{3} \rfloor = \lfloor \frac{5}{3} \rfloor = 1$ and $\lfloor \frac{s_6.T_E - \mathit{off}}{d} \rfloor = \lfloor \frac{\text{2012-10} - \text{2012-1}}{3} \rfloor = \lfloor \frac{9}{3} \rfloor = 3$. Five out of ten partitions are empty, namely $p_{0,3}$, $p_{0,2}$, $p_{1,2}$, $p_{2,2}$, and $p_{3,3}$.

Lemma 2 ($\mathcal{OIP}$ Overlap Query)**.** *Let (k, d, off) be an $\mathcal{OIP}$ configuration and $Q = [Q_S, Q_E]$ be a query interval. The candidate tuples that possibly overlap Q are in partitions $p_{i,j}$ for which $i \leq e = \lfloor \frac{Q_E - \mathit{off}}{d} \rfloor$ and $j \geq s = \lfloor \frac{Q_S - \mathit{off}}{d} \rfloor$. We term these partitions the* relevant partitions*; s is the start and e the end index of Q.*

Proof. (By contradiction) Assume a partition $p_{i,j}$ with $j < s = \lfloor \frac{Q_S - \mathit{off}}{d} \rfloor$, which contains a tuple r that overlaps Q. According to Definition 14, tuple r is placed in a partition $p_{i,j}$ with $i = \lfloor \frac{r.T_S - \mathit{off}}{d} \rfloor$ and $j = \lfloor \frac{r.T_E - \mathit{off}}{d} \rfloor$. Since $j < s$, we get $r.T_E < Q_S$, i.e., $r.T$ and Q do not overlap, which contradicts our assumption. Similarly, a tuple in $p_{i,j}$ with $i > e = \lfloor \frac{Q_E - \mathit{off}}{d} \rfloor$ cannot overlap Q since $r.T_S > Q_E$. □

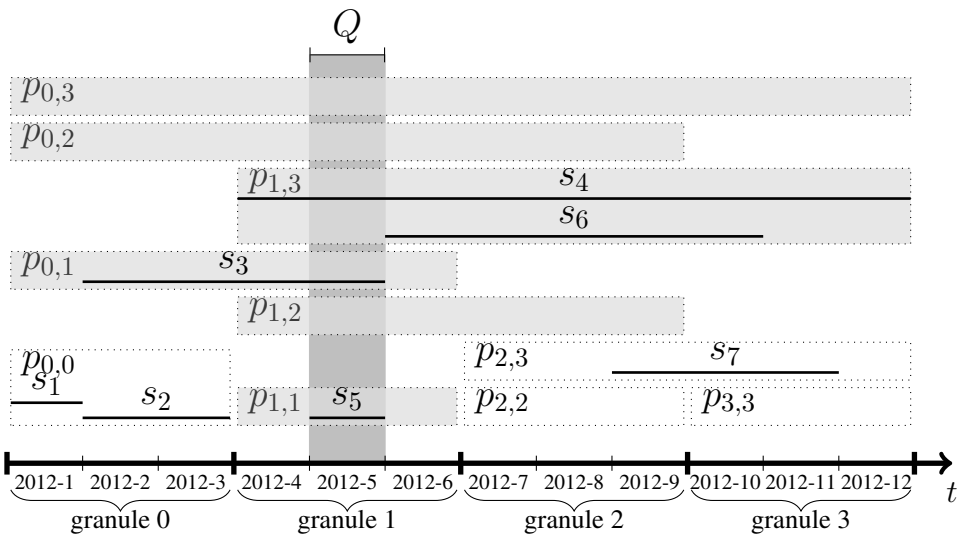


Figure 4.2: $\mathcal{OIP}$ with Configuration $(4, 3, \text{2012-1})$ for $\mathbf{s}$.

Example 17. Consider Figure 4.2 with the $\mathcal{OIP}$ configuration $(4, 3, \text{2012-1})$ and the query interval $Q = [\text{2012-5}, \text{2012-5}]$. For the relevant partitions, $p_{i,j}$, the following constraints hold: $i \leq e = \lfloor \frac{Q_E - off}{d} \rfloor = \lfloor \frac{\text{2012-5} - \text{2012-1}}{3} \rfloor = \lfloor \frac{4}{3} \rfloor = 1$ and $j \geq s = \lfloor \frac{Q_S - off}{d} \rfloor = \lfloor \frac{\text{2012-5} - \text{2012-1}}{3} \rfloor = \lfloor \frac{4}{3} \rfloor = 1$. This is satisfied for partitions $p_{0,3}$, $p_{0,2}$, $p_{0,1}$, $p_{1,3}$, $p_{1,2}$, and $p_{1,1}$ (gray boxes), which contain all candidate tuples for the query interval Q.

Next, we establish the *constant clustering guarantee* of $\mathcal{OIP}$: the difference in duration between a tuple and its partition is less than two granules, i.e., constant and independent of the duration of a tuple. Note that the number of time points per granule or the duration of a granule have no impact on our solution. The constant clustering guarantee ensures (a) an excellent partitioning since the difference in duration between a tuple and its partitions is less than two granules, (b) an average false hit ratio that is independent of the intervals of tuples (cf. Section 4.5.1), and (c) it allows to take advantage of empty partitions by increasing k (cf. Section 4.6.2).

Lemma 3 (Constant Clustering Guarantee)**.** *Let (k, d, off) be an $\mathcal{OIP}$ configuration for relation $\mathbf{r}$. The difference in duration between a tuple $r \in \mathbf{r}$ and its partition p is less than $2d$:*

$$\forall r \in \mathbf{r}(r \in p \Rightarrow |p.T| - |r.T| < 2d).$$

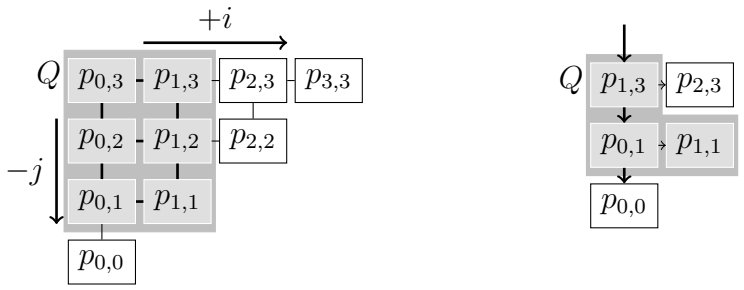
Proof. We show that the difference in duration of the smallest tuple in a partition $p_{i,j}$ and $p_{i,j}$ is less than $2d$. A tuple r is placed in partition $p_{i,j}$ iff $i = \lfloor \frac{r.T_S - off}{d} \rfloor$ and $j = \lfloor \frac{r.T_E - off}{d} \rfloor$ (cf. Definition 14). Thus, we have $i \cdot d \leq r.T_S - off \leq (i+1)d - 1$ and $j \cdot d \leq r.T_E - off \leq (j+1)d - 1$.

The smallest tuple in $p_{i,j}$ has duration $|[(i+1)d-1, j \cdot d]| = j \cdot d - (i+1)d + 2$, and $p_{i,j}$ has duration $|[i \cdot d, (j+1)d-1]| = (j+1)d - i \cdot d$. Hence, the difference in duration between the smallest tuple r in $p_{i,j}$ and $p_{i,j}$ is $2d - 2 < 2d$. □

For instance, for a relation that spans 2000 days and with $k = 200$, we have $d = 10$ days. A tuple that spans 80 days can be in a partition that spans 90 days, which is a difference of 10 days. A tuple that spans 282 days can be in a partition that spans 300 days, which is a difference of 18 days. Note that the difference is always less than $2d = 20$ days.

4.4.2 Lazy Partitioning

We represent the $\mathcal{OIP}$ access structure as a triangle in a distance regular square grid graph [BCN89], as illustrated in Figure 4.3a for the $\mathcal{OIP}$ in Figure 4.2. We call this a *triangular grid graph* with grid points (i, j) for $0 \leq i \leq j < k$. To find all relevant partitions for a query interval with start index s and end index e, we determine all partitions $p_{i,j}$ for which $j \geq s$ and $i \leq e$ (cf. Lemma 2). We start at the top-left corner of the grid (i.e., $i = 0$, $j = k - 1 = 3$) and move along the path with decreasing j as long as $j \geq s$. At each node $p_{0,j}$, we follow the path with increasing i as long as $i \leq min(j, e)$. In Figure 4.3a, the relevant partitions (gray) for query interval Q are on the paths $p_{0,3} \rightarrow p_{1,3}$, $p_{0,2} \rightarrow p_{1,2}$ and $p_{0,1} \rightarrow p_{1,1}$.



(a) Triangular Grid Graph. (b) Lazy Partition List.

Figure 4.3: Management of $\mathcal{OIP}$ Partitions.

The number of possible $\mathcal{OIP}$ partitions corresponds to the number of nodes in a triangular grid graph and grows quadratically with the number of granules k.

Proposition 5 (Number of Partitions)**.** *For k granules, the number of possible partitions is* $|\mathbf{p}| = \frac{k^2+k}{2}$.

Proof. There are $(k-i)$ partitions that start together with the ith granule, for $0 \leq i < k$, which yields $|\mathbf{p}| = \sum_{i=0}^{k-1}(k-i) = \sum_{i=0}^{k-1}(k) - \sum_{i=0}^{k-1}(i) = \frac{k^2+k}{2}$. □

The *lazy partition list* is a compressed triangular grid graph that includes only non-empty partitions. Figure 4.3b shows the lazy partition list of our example. It includes only the non-empty partitions and the directed edges that are needed for navigation. The *main list* starts at the upper-left corner and connects nodes with decreasing j from top to bottom. Each node of the main list starts a *branch list* that connects (from left to right) nodes with the same j-value and increasing i-value.

The lazy partition list has the following advantages: (a) the number of CPU operations is reduced since empty partitions do not appear in the access structure; (b) k can be increased if not all partitions are used (cf. Section 4.5.2 and Section 4.6.2); and (c) the number of partitions is upper bounded by the cardinality of the partitioned relation, independent of the value of k.

Lemma 4 (Number of Partitions with Lazy Partitioning)**.** *Assume an $\mathcal{OIP}$ configuration (k, d, off) for a relation $\mathbf{r}$ with n tuples whose valid time duration is at most λ. The number of partitions, $|\mathbf{p}|$, of $\mathcal{OIP}$ for $\mathbf{r}$ is upper bounded by $min(k\lceil\lambda \cdot k\rceil + k - \frac{\lceil\lambda\cdot k\rceil^2}{2} - \frac{\lceil\lambda\cdot k\rceil}{2}, n)$.*

Proof. Tuples in $\mathbf{r}$ span at most $\lceil\lambda \cdot k\rceil$ granules. From Lemma 3 we have that the difference in duration of a partition and its tuples is less than two granules. Thus, the longest used partition spans at most $\lceil\lambda \cdot k\rceil + 1$ granules, and we have $|\mathbf{p}| \leq \sum_{x=0}^{\lceil\lambda\cdot k\rceil+1-1}(k-x) = k\lceil\lambda \cdot k\rceil + k - \frac{\lceil\lambda\cdot k\rceil^2}{2} - \frac{\lceil\lambda\cdot k\rceil}{2}$. Since empty partitions are not created, $|\mathbf{p}|$ cannot exceed n. □

Example 18. Assume a relation with tuple durations up to 20% of the relation's time range, i.e., $\lambda = 0.2$. With $k = 200$, at most $200\lceil 0.2 \cdot 200\rceil + 200 - \frac{\lceil 0.2\cdot 200\rceil^2}{2} - \frac{\lceil 0.2\cdot 200\rceil}{2} = 7,380$ partitions out of $\frac{200^2+200}{2} = 20,100$ possible partitions are used, i.e., 37%, while 63% are empty.

4.4.3 Implementation of $\mathcal{OIP}$

Our implementation of $\mathcal{OIP}$ uses a lazy partition list, $\mathcal{L}$, to keep track of indices and storage blocks of partitions. Figure 4.4 shows the lazy partition list for our running example.

Algorithm 2 creates the lazy partition list $\mathcal{L}$ for an input relation $\mathbf{r}$ with n tuples and an $\mathcal{OIP}$ configuration (k, d, off). After initializing an empty partition list, the relation is sorted according to the tuples' partition $p_{i,j}$ with j in ascending and i in descending order. The indices i and j of

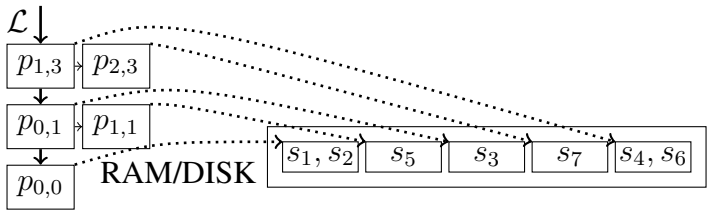


Figure 4.4: $\mathcal{OIP}$ Lazy Partition List $\mathcal{L}$ with Block Pointers.

the partition to which a tuple r is assigned are computed according to Definition 14. The sorting ensures that tuples fall either into the first node of the list ($c = nil \vee c.j < j$) or into a new node that is prepended to $\mathcal{L}.head$ ($c.i > i$). The sorting reduces the complexity of insertions from $O(k)$ to $O(1)$ and gives a total runtime complexity for constructing $\mathcal{L}$ of $O(n \log n)$, which is independent of k and ensures that storage blocks are allocated sequentially.

Algorithm 2: OIPCREATE (**r**, (k,d,off))

Input: Relation **r** and $\mathcal{OIP}$ configuration (k, d, off)
Output: Partition list $\mathcal{L}$

```
L := empty partition list;
Sort r by ⌊(r.T_E − off)/d⌋ ASC and ⌊(r.T_S − off)/d⌋ DESC;
foreach r ∈ r do
    i := ⌊(r.T_S − off)/d⌋;
    j := ⌊(r.T_E − off)/d⌋;
    c := L.head;
    if c = nil ∨ c.j < j then
        L.head := new node with partition p_{i,j};
        L.head.down := c;
    else if c.i > i then
        L.head := new node with partition p_{i,j};
        L.head.down := c.down;
        L.head.right := c;
    Add r to L.head;
return L;
```

Example 19. Consider relation s in Figure 4.2. The call of OIPCREATE (s, (4,3,2012-1)) constructs $\mathcal{L}$[2] as follows:

1. $\mathbf{r} = sort(\mathbf{s}) = \langle s_1, s_2, s_5, s_3, s_7, s_4, s_6 \rangle$, $\mathcal{L} = \langle\rangle$

[2]We use nested lists as an abstract notation. For instance, $\mathcal{L} = \langle\langle a \rangle, \langle b, c \rangle\rangle$ has nodes a, b, c; $\mathcal{L}.head = a$; $a.down = b$; $b.right = c$; $a.right$, $b.down$, $c.down$, and $c.right$ are nil.

2. $r = s_1$, $i = \lfloor \frac{\text{2012-1}-\text{2012-1}}{3} \rfloor = 0$, $j = \lfloor \frac{\text{2012-1}-\text{2012-1}}{3} \rfloor = 0$,
   $\mathcal{L} = \langle\langle(0, 0, \{s_1\})\rangle\rangle$

3. $r = s_2$, $i = \lfloor \frac{\text{2012-2}-\text{2012-1}}{3} \rfloor = 0$, $j = \lfloor \frac{\text{2012-3}-\text{2012-1}}{3} \rfloor = 0$,
   $\mathcal{L} = \langle\langle(0, 0, \{s_1, s_2\})\rangle\rangle$

4. $r = s_5$, $i = \lfloor \frac{\text{2012-5}-\text{2012-1}}{3} \rfloor = 1$, $j = \lfloor \frac{\text{2012-5}-\text{2012-1}}{3} \rfloor = 1$,
   $\mathcal{L} = \langle\langle(1, 1, \{s_5\})\rangle, \langle(0, 0, \{s_1, s_2\})\rangle\rangle$

5. $r = s_3$, $i = \lfloor \frac{\text{2012-2}-\text{2012-1}}{3} \rfloor = 0$, $j = \lfloor \frac{\text{2012-5}-\text{2012-1}}{3} \rfloor = 1$,
   $\mathcal{L} = \langle\langle(0, 1, \{s_3\}), (1, 1, \{s_5\})\rangle, \langle(0, 0, \{s_1, s_2\})\rangle\rangle$

The algorithm terminates and returns the lazy partition list $\mathcal{L} = \langle\langle(1, 3, \{s_4, s_6\}), (2, 3, \{s_7\})\rangle, \langle(0, 1, \{s_3\}), (1, 1, \{s_5\})\rangle, \langle(0, 0, \{s_1, s_2\})\rangle\rangle$, which is illustrated in Figure 4.4.

4.5 Analytical Results of $\mathcal{OIP}$

In this section, we analyze the quality of $\mathcal{OIP}$ using two different measures. The *average false hit ratio*, AFR, measures the precision of a partitioning in terms of the average number of tuples that are retrieved for a query interval but do not contribute to the result. The *average number of partition accesses*, APA, quantifies the number of partitions that are fetched for a query.

4.5.1 Average False Hit Ratio

Definition 15. (*False Hits*) Let P be a partitioning of a valid-time relation $\mathbf{r}$ and Q be a query interval. The *false hits*, $\mathsf{F}(P, Q)$, are the tuples that are retrieved when fetching the relevant partitions, but are not part of the query result, i.e.,

$$\mathsf{F}(P, Q) = \{r \mid \exists p \in P\big(r \in p \wedge p.T \cap Q \wedge \neg\,(r.T \cap Q)\big)\}.$$

Consider Figure 4.2. For the query interval $Q = [\text{2012-5}, \text{2012-5}]$, only the relevant partitions are fetched (i.e., $p_{1,1}$, $p_{0,1}$, $p_{1,3}$). The false hits are $\mathsf{F}(\mathcal{OIP}, Q) = \{s_6\}$ since partition $p_{1,3}$ is fetched, but s_6 does not overlap Q. The result tuples are s_3, s_4, and s_5.

We proceed by defining the sum false hit ratio as the percentage of false hits over all possible point queries, i.e., the false hits produced by all queries over query intervals of duration 1 divided by the total number of tuples.

Definition 16. (*Sum False Hit Ratio*) Let P be a partitioning of a valid-time relation $\mathbf{r}$ with time range U. The *sum false hit ratio*, $\mathsf{SFR}(P)$, for all query intervals $[x, x]$ that overlap U is defined as

$$\mathsf{SFR}(P) = \frac{\sum_{x \in U} |\mathsf{F}(P, [x, x])|}{|\mathbf{r}|}.$$

For the $\mathcal{OIP}$ shown in Figure 4.2, we have $\mathsf{SFR}(\mathcal{OIP}) = \frac{|\mathsf{F}(\mathcal{OIP},[2012\text{-}1,2012\text{-}1])|+\cdots+|\mathsf{F}(\mathcal{OIP},[2012\text{-}12,2012\text{-}12])|}{7} = \frac{14}{7} = 2$. Thus, for all query intervals of duration 1, two times the number of tuples in $\mathbf{s}$ are retrieved as false hits.

Lemma 5. *The sum false hit ratio of a partitioning P over a time range U is independent of the query interval duration q, i.e., it is the same for all query intervals $[x, x + q - 1]$ that overlap U for any value of q:*

$$\mathsf{SFR}(P) = \frac{\sum_{x \in U} |\mathsf{F}(P, [x, x])|}{|\mathbf{r}|} = \frac{\sum_{Q:Q \cap U \wedge |Q|=q} |\mathsf{F}(P, Q)|}{|\mathbf{r}|}.$$

Proof. Consider a time point $x \in U$ and a partition $p \in P$. Query interval $[x, x]$ of duration 1 can produce false hits in p if there is a non-overlapping part before x (i.e., $p.T_S < x$) and/or a non-overlapping part after x (i.e., $x < p.T_E$). All tuples that start and end in one of the two non-overlapping parts are false hits. We consider now query intervals of duration $q > 1$. The query interval $[x, x+q-1]$ produces the same non-overlapping part before x, and the query interval $[x-q+1, x]$ the same non-overlapping part after x, yielding together exactly the same false hits for partition p as the point query with interval $[x, x]$. Since for each x there exists exactly one query interval of duration q that starts at x and one that ends at x, it is straightforward to generalize this result to the sum over all partitions and time points in U. This proves the lemma. □

Next, we define the average false hit ratio for an arbitrary query interval of duration $q \geq 1$.

Definition 17. (*Average False Hit Ratio*) Let P be a partitioning for a relation $\mathbf{r}$ with time range U. The *average false hit ratio*, $\mathsf{AFR}(P)$, for a query interval duration q is defined as

$$\mathsf{AFR}(P) = \frac{\mathsf{SFR}(P)}{|U| + q - 1}.$$

Proposition 6. *The* $\mathsf{AFR}(P)$ *decreases monotonically with increasing query interval duration* q*.*

Example 20. Consider Figure 4.2 with the time range $U = [\text{2012-1}, \text{2012-12}]$ and the sum false hit ratio $\mathsf{SFR}(\mathcal{OIP}) = 2$. The number of query intervals of duration $q = 1$ is $|U| + q - 1 = 12$, yielding an average false hit ratio $\mathsf{AFR}(\mathcal{OIP}) = 2 \cdot \frac{1}{12}$ $(= 16.7\%)$, i.e., on average 16.7% of the fetched tuples are false hits. The number of query intervals of duration $q = 5$ is $|U| + q - 1 = 16$, yielding an average false hit ratio $\mathsf{AFR}(\mathcal{OIP}) = 2 \cdot \frac{1}{16}$ $(= 12.5\%)$.

For the analysis of the average false hit ratio of $\mathcal{OIP}$ in the following Theorem 2, we use duration complete relations. A *duration complete relation*, $\mathbf{r}^l_U$, contains exactly one tuple for each interval up to a duration l in the time range U, i.e.,

$$\begin{aligned}
&\forall T \subseteq U(|T| \leq l \Rightarrow \exists r \in \mathbf{r}^l_U(r.T = T)),\\
&\forall r \in \mathbf{r}^l_U(|r.T| \leq l),\\
&\forall r, r' \in \mathbf{r}^l_U(r \neq r' \Rightarrow r.T \neq r'.T).
\end{aligned}$$

For instance, the duration complete relation $\mathbf{r}^2_{[0,3]}$ contains a total of seven tuples with intervals $[0,0]$, $[1,1]$, $[2,2]$, $[3,3]$, $[0,1]$, $[1,2]$, $[2,3]$. Duration complete relations ensure that the AFR is calculated over tuples of all possible positions and durations in U.

Theorem 2. *Assume an* $\mathcal{OIP}$ *with configuration* (k, d, off)*. The average false hit ratio for duration complete relations is independent of the duration of the tuples and upper bounded by*

$$\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}.$$

Proof. The proof is split into three parts. In Part 1 we compute the SFR of $\mathcal{OIP}$ for a duration complete relation $\mathbf{r}^l_{kd}$ and a query interval with duration q. In Part 2 we use the result of Part 1 to show that $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$ for tuples of duration $l = 1$ and query intervals of duration $q = 1$. From Proposition 6 we have that the $\mathsf{AFR}(\mathcal{OIP})$ for $q > 1$ is smaller. In Part 3 we show that for tuples up to larger durations, i.e., $l > 1 \leq k \cdot d$, the SFR of $\mathcal{OIP}$ and the $\mathsf{AFR}(\mathcal{OIP})$ are smaller than for $l = 1$, thus $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$.

Part 1: Assume $l \leq d$. We compute first the sum of false hits for partitions p that span one granule. Assume a query interval Q with $q = 1$ that overlaps p. Let $v \geq 0$ be the duration of the non-overlapping part of p before Q starts. If $v \leq l$, we have $\frac{v^2+v}{2}$ false hits in this part of the partition, i.e., all intervals in p that start and end before Q. If $v \geq l$, we have $\frac{v^2+v}{2} - \sum_{x=l}^{v}(v-x) = vl - \frac{l^2-l}{2}$ false hits in this part, i.e., all intervals in p up to duration l that start and end before Q. We sum the false hits of all query intervals of duration 1 and get $\sum_{v=0}^{l-1}(\frac{v^2+v}{2}) + \sum_{v=l}^{d-1}(vl - \frac{l^2-l}{2}) = \frac{l^3-l}{6} + \frac{ld^2-l^2d}{2}$. The same sum is obtained for false hits in the non-overlapping part after the query interval Q ends. Thus, for k granules we get a total of $k(\frac{l^3-l}{3} + ld^2 - l^2d)$ false hits.

Next, we compute the sum of false hits for partitions that span more than one granule. For $l \leq d$, only partitions of duration $2d$ contain tuples. Each of these partitions contains $\frac{l^2-l}{2}$ tuples, i.e., all tuples up to duration l that start in the first half and end in the second half of the partition. The total number of matching tuples for all query intervals of duration 1 is $2\sum_{p=1}^{l-1}\sum_{x=1}^{p}(x) = \frac{l^3-l}{3}$. Thus, for $k-1$ partitions of duration $2d$ there are $2d$ possible query intervals of duration $q = 1$ that overlap a partition with $\frac{l^2-l}{2}$ tuples. Subtracting from these tuples the $\frac{l^3-l}{3}$ matches gives a total of $(k-1)(2d\frac{l^2-l}{2} - \frac{l^3-l}{3})$ false hits.

Adding up the false hits for the partitions and dividing the sum by the number of tuples $|\mathbf{r}^l_{kd}| = kdl - \frac{l^2-l}{2}$ we get

$$\mathsf{SFR}(\mathcal{OIP}) = \frac{2(l^2 - 3dl + 3kd^2 - 3kd + 3d - 1)}{3(2kd - l + 1)} \text{ for } l \leq d. \tag{4.1}$$

Now assume $l > d$. Let l be a multiple of d and $h = l/d$. Partitions that span one granule are completely full. This yields a total of $k\frac{d^3-d}{3}$ false hits, by replacing l in the first case of the proof with d. Then we have $\sum_{x=1}^{h-1}(k-x)$ partitions that span more than one granule, are not longer than l, and are completely filled. Each of these partitions produces up to $d^3 - d^2$ false hits. The only partitions that are longer than l and contain tuples of duration $l = hd$ have duration $d(h+1)$, of which $(k-h)$ exist. Each of these partitions contains $\frac{d^2-d}{2}$ tuples up to size l. The total number of matching tuples for all query intervals of duration 1 is $2\sum_{p=1}^{d-1}\sum_{x=1}^{p}(x) = \frac{d^3-d}{3}$, and the total number of matches where no false hits are possible is $\frac{d(h-1)(d^2-d)}{2}$. Thus, for $k-h$ partitions we get $(k-h)\Big(d(h+1)\frac{d^2-d}{2} - (\frac{d^3-d}{3} + \frac{d(h-1)(d^2-d)}{2})\Big)$. Finally, we divide the sum by the number of tuples $|\mathbf{r}^l_{kd}| = kdl - \frac{l^2-l}{2}$ and get

$$\mathsf{SFR}(\mathcal{OIP}) = \frac{(d-1)(6kd-d+2-3l)}{3(2kd-l+1)} \text{ for } l > d. \tag{4.2}$$

Part 2: We show that $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$ for tuples of duration $l = 1$ and query intervals of duration $q = 1$. We use the $\mathsf{SFR}(\mathcal{OIP})$ of Part 1 for $l \leq d$ (since d must be at least 1) and set $l = 1$ in Equation (4.1) to get:

$$\mathsf{SFR}(\mathcal{OIP}) = \frac{2(1^2 - 3d1 + 3kd^2 - 3kd + 3d - 1)}{3(2kd - 1 + 1)} = d - 1$$

Next, we set $q = 1$ and divide by the number of query intervals $kd + q - 1 = kd + q - 1 = kd$ (ref. Definition 17), and get

$$\mathsf{AFR}(\mathcal{OIP}) = \frac{\mathsf{SFR}(\mathcal{OIP})}{kd+q-1} = \frac{d-1}{kd+1-1} = \frac{1}{k} - \frac{1}{kd} < \frac{1}{k}.$$

Part 3: We show that for $l > 1 \leq kd$ the $\mathsf{SFR}(\mathcal{OIP})$ is smaller than for $l = 1$, i.e., smaller $d-1$ and thus $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$. Recall that the SFR is independent of the query interval duration q (ref. Lemma 5). First, we consider $1 < l \leq d$ and Equation (4.1):

$$\begin{aligned} d - 1 &> \frac{2(l^2 - 3dl + 3kd^2 - 3kd + 3d - 1)}{3(2kd - l + 1)} \\ 3(2kd - l + 1)(d - 1) &> 2(l^2 - 3dl + 3kd^2 - 3kd + 3d - 1) \\ -2l^2 + (3 + 3d)l - 3d - 1 &> 0 \end{aligned}$$

By solving the quadratic equation $-2l^2 + (3 + 3d)l - 3d - 1 = 0$ we get $l = \{1, \frac{3d+1}{2}\}$, since the quadratic term is negative we have an concave down parabola and thus the inequality we need to show for $1 < l \leq d$ holds for $1 < l < \frac{3d+1}{2}$.

Second, we consider $d < l \leq kd$ and Equation (4.2):

$$
\begin{aligned}
d - 1 &> \frac{(d-1)(6kd - d + 2 - 3l)}{3(2kd - l + 1)} \\
3(2kd - l + 1)(d-1) &> (d-1)(6kd - d + 2 - 3l) \\
6kd - 3l + 3 &> 6kd - d + 2 - 3l \\
d &> -1
\end{aligned}
$$

Since $d \geq 1$, we have that the $\mathsf{SFR}(\mathcal{OIP})$ and the $\mathsf{AFR}(\mathcal{OIP})$ for $l > 1$ is smaller than for $l = 1$, thus $\mathsf{AFR}(\mathcal{OIP}) < \frac{1}{k}$. □

4.5.2 Average Number of Partition Accesses

The *average number of partition accesses*, APA, quantifies how many partitions are accessed on average to retrieve all tuples that overlap a query interval, i.e., how many relevant partitions exist.

Lemma 6 (APA Upper Bound)**.** *Assume an $\mathcal{OIP}$ with configuration (k, d, off), where all partitions are used. The average number of partition accesses for query intervals with uniformly distributed start and end time points is:*

$$\mathsf{APA}(\mathcal{OIP}) \leq \frac{k^2 + k + 1}{3}.$$

Proof. For query intervals with uniformly distributed start and end time points, every query interval starting in granule s and ending in granule e has the same probability. Thus, we need to compute the number of partitions that a query interval accesses when starting in s and ending in e, which is the total number of partitions minus all partitions ending before s and all partitions starting after e as follows:

$$
\begin{aligned}
\#acc(s,e) &= \frac{k^2 + k}{2} - \sum_{i=0}^{s-1}(s - i) - \sum_{i=0}^{k-e-1}(k - e - 1 - i) \\
&= k + k \cdot e - \frac{s^2 + s}{2} - \frac{e^2 + e}{2}.
\end{aligned}
$$

We sum the number of partition accesses, $\#acc(s,e)$, for all $s \leq e < k$ and divide the sum by the cardinality of $s \leq e < k$ to get:

$$\mathsf{APA}(\mathcal{OIP}) = \frac{\sum_{e=0}^{k-1}\sum_{s=0}^{e}(\#acc(s,e))}{\sum_{e=0}^{k-1}\sum_{s=0}^{e}(1)} = \frac{k^2+k+1}{3}.$$

□

Since empty partitions are not present in the $\mathcal{OIP}$ access structure, APA is reduced if not all partitions are used. We use a tightening factor to quantify the reduction of partitions with lazy partitioning. The *tightening factor*, τ, with $0 < \tau \leq 1$, is calculated as the ratio between the number of used partitions with lazy partitioning (Lemma 4) and the total number of partitions (Proposition 5). For instance, the tightening factor in Example 18 is $\tau = 1890/5050 = 0.37$.

Theorem 3 (APA)**.** *Assume an $\mathcal{OIP}$ configuration (k, d, off) for a relation with n tuples and a tightening factor τ with $0 < \tau \leq 1$. The average number of partition accesses is:*

$$\mathsf{APA}(\mathcal{OIP}) \leq min(\tau \cdot \frac{k^2+k+1}{3}, n).$$

Proof. The proof follows from Lemma 6 and Lemma 4. The tightening factor τ is the ratio of the number of used and the number of possible partitions. The inequality holds since the multiplication with τ conservatively assumes that the longest partitions, which produce more partition accesses than shorter partitions, are omitted. □

4.6 The Overlap Join OIPJOIN

This section presents the OIPJOIN algorithm, derives the optimal number of granules k, illustrates its calculation by an example, and analyzes the runtime complexity of the OIPJOIN.

4.6.1 The OIPJOIN Algorithm

Algorithm 3 computes the OIPJOIN for relations **r** and **s**. First, k (cf. Section 4.6.2) and the $\mathcal{OIP}$ configurations of the two relations are determined. The partitions are created by calling OIPCREATE, and the result relation **z** is initialized. Then, the algorithm iterates over each outer

partition in $\mathcal{L}_r$ and performs an overlap query (cf. Lemma 2) with the query interval $[Q_S, Q_E]$ of the outer partition (cf. Definition 14). If $[Q_S, Q_E]$ does not overlap the time range of the inner relation $\mathbf{s}$, the outer partition is skipped. Otherwise, the indices s and e of the query interval $[Q_S, Q_E]$ are determined. The relevant partitions of the inner relation that overlap with the query interval are fetched, and the tuples are joined with the tuples in the outer partition. The result tuples are collected in $\mathbf{z}$.

Algorithm 3: OIPJOIN ($\mathbf{r}$, $\mathbf{s}$)

Input: Relation $\mathbf{r}$ and relation $\mathbf{s}$
Output: $\mathbf{z} = \{r \circ s \mid r \in \mathbf{r} \wedge s \in \mathbf{s} \wedge r.T \cap s.T\}$

Determine k for $\mathbf{r}$ and $\mathbf{s}$ for given IO and CPU costs;
Determine configurations (k, d_r, off_r) for $\mathbf{r}$ and (k, d_s, off_s) for $\mathbf{s}$;
$\mathcal{L}_r \leftarrow$ OIPCREATE($\mathbf{r}$, (k, d_r, off_r));
$\mathcal{L}_s \leftarrow$ OIPCREATE($\mathbf{s}$, (k, d_s, off_s));
$\mathbf{z} := \emptyset$;
foreach *node* c_r *in* $\mathcal{L}_r$ **do**
  $Q_S := \mathit{off}_r + c_r.i \cdot d_r$;
  $Q_E := \mathit{off}_r + (c_r.j + 1) \cdot d_r - 1$;
  if $Q_E \geq \mathit{off}_s \wedge Q_S < \mathit{off}_s + k \cdot d_s$ **then**
    $s := \lfloor \frac{Q_S - \mathit{off}_s}{d_s} \rfloor$;
    $e := \lfloor \frac{Q_E - \mathit{off}_s}{d_s} \rfloor$;
    $c_s := \mathcal{L}_s.\mathit{head}$;
    while $c_s \neq \mathit{nil} \wedge c_s.j \geq s$ **do**
      $x := c_s$;
      while $x \neq \mathit{nil} \wedge x.i \leq e$ **do**
        $\mathbf{z} := \mathbf{z} \cup \{$ joined tuples from c_r and $x\}$;
        $x := x.\mathit{right}$;
      $c_s := c_s.\mathit{down}$;
return $\mathbf{z}$;

Example 21. Consider Figure 4.1 with $k = 4$. We get the $\mathcal{OIP}$ configurations $(4, 2, \text{2012-5})$ for $\mathbf{r}$ and $(4, 3, \text{2012-1})$ for $\mathbf{s}$. OIPCREATE creates the lazy partition lists $\mathcal{L}_r = \langle\langle(1, 3, \{r_3\})\rangle, \langle(0, 0, \{r_1, r_2\})\rangle\rangle$ and $\mathcal{L}_s = \langle\langle(1, 3, \{s_4, s_6\}), (2, 3, \{s_7\})\rangle, \langle(0, 1, \{s_3\}), (1, 1, \{s_5\})\rangle, \langle(0, 0, \{s_1, s_2\})\rangle\rangle$. The first outer partition is processed as follows:

$$c_r = (1, 3, \{r_3\})$$
$$Q_S = \text{2012-5} + 1 \cdot 2 = \text{2012-7}$$

$$
\begin{aligned}
Q_E &= \text{2012-5} + (3+1) \cdot 2 - 1 = \text{2012-12} \\
s &= \lfloor \tfrac{\text{2012-7} - \text{2012-1}}{3} \rfloor = \lfloor \tfrac{6}{3} \rfloor = 2 \\
e &= \lfloor \tfrac{\text{2012-12} - \text{2012-1}}{3} \rfloor = \lfloor \tfrac{11}{3} \rfloor = 3 \\
c_s &= \mathcal{L}_s.head = (1, 3, \{s_4, s_6\}) \\
x &= c_s = (1, 3, \{s_4, s_6\}) \\
\mathbf{z} &= \{r_3 \circ s_4, r_3 \circ s_6\} \\
x &= x.right = (2, 3, \{s_7\}) \\
\mathbf{z} &= \{r_3 \circ s_4, r_3 \circ s_6, r_3 \circ s_7\} \\
c_s &= c_s.down = (0, 1, \{s_3\})
\end{aligned}
$$

The second (and last) outer partition, $c_r = (0, 0, \{r_1, r_2\})$, is processed in a similar way, yielding the final result $\mathbf{z} = \{r_3 \circ s_4, r_3 \circ s_6, r_3 \circ s_7, r_1 \circ s_3, r_1 \circ s_4, r_1 \circ s_5, r_2 \circ s_4, r_2 \circ s_6\}$.

4.6.2 Number of Granules k

Choosing k (i.e., the number of granules) is the most important decision for the OIPJOIN. In order to derive k for the outer and the inner relation, we proceed in two steps. First, we provide a cost function for the OIPJOIN, and second, we minimize the cost function with respect to k.

Cost Function

The cost function considers the CPU cost of a comparison operation (c_cpu) and the cost of a block IO (c_io). A block IO can refer to either main memory or disk. The cost function models the *overhead* due to partition accesses and false hits. Recall that the cost for creating the partitioning is, due to sorting, independent of k and thus not included in the cost function.

Let k_r and k_s be the number of granules for the outer and inner relation, respectively. For the join we fetch, for each of the $O(k_r^2)$ outer partitions, $O(k_s^2)$ inner partitions, i.e., $O(k_r^2 \cdot k_s^2)$. Furthermore, for each outer and inner tuple we have, respectively, $O(\frac{n_s}{k_s})$ and $O(\frac{n_r}{k_r})$ false hits, i.e., $O(n_s \cdot \frac{n_r}{k_r} + n_r \cdot \frac{n_s}{k_s})$. Both, $O(k_r^2 \cdot k_s^2)$ and $O(n_s \cdot \frac{n_r}{k_r} + n_r \cdot \frac{n_s}{k_s})$ reach their minimum when $k_r = k_s$, i.e., outer and inner relation are partitioned using the same number of granules k. Thus,

we have a cost function with $k = k_r = k_s$:

$$\begin{aligned} cost(k) &= |\mathbf{p}_r| \cdot \mathsf{APA} \cdot (c\_io + 2 \cdot c\_cpu) + \\ &\quad |\mathbf{p}_r| \cdot n_s \cdot \mathsf{AFR} \cdot (\frac{c\_io}{b} + 2 \cdot \frac{n_r}{|\mathbf{p}_r|} \cdot 2 \cdot c\_cpu) \\ &= x \cdot \mathsf{APA} + y \cdot \mathsf{AFR} \end{aligned} \tag{4.3}$$

The first line is the cost for partition accesses. For each of the $|\mathbf{p}_r|$ outer partitions, the algorithm accesses APA inner partitions. Each partition access costs one extra c_io since an inner partition can have at most one partially filled block (remember we only measure the overhead) and two c_cpu (comparison i and j) for checking if this partition in the lazy partition list is relevant. The second line is the cost for false hits. For each of the $|\mathbf{p}_r|$ outer partitions, $n_s \cdot \mathsf{AFR}$ false hits in the relevant inner partitions produce extra block transfers, where b is the average number of tuples per block of the inner relation. The costs for identifying false hits is two c_cpu (comparison T_S and T_E) for each false hit in the outer partition and each false hit in the relevant inner partitions.

Determining k

We derive k by minimizing the cost function using the partial derivative of $x \cdot \mathsf{APA} + y \cdot \mathsf{AFR}$. The terms quantify, respectively, the increase of the costs due to partition accesses and the decrease of the costs due to false hits. k can be increased as long as the costs for AFR decrease more than the costs for APA increase. The optimal k is the point where the cost for accessing partitions starts growing faster than the cost for false hits decreases, which is the minimum of the cost function.

Since the complexity of the cost function prevents an analytical solution of the minimization problem, we proceed in two steps to derive k. First, we keep $|\mathbf{p}_r|$ and τ constant and derive k as follows:

1. Compute the partial derivative of $x \cdot \mathsf{APA} + y \cdot \mathsf{AFR}$. We use APA and AFR from Theorems 2 and 3 to get $x \cdot \tau \cdot \frac{k^2+k+1}{3} + y \cdot \frac{1}{k}$. The partial derivative with respect to k is $\partial_k(x \cdot \tau \cdot \frac{k^2+k+1}{3} + y \cdot \frac{1}{k}) = x \cdot \tau \cdot (\frac{2}{3} \cdot k + \frac{1}{3}) - \frac{y}{k^2}$.

2. Solve $x \cdot \tau \cdot (\frac{2}{3} \cdot k + \frac{1}{3}) - \frac{y}{k^2} = 0$ to get the k that minimizes the cost function:

$$k = \frac{\sqrt[3]{(162 \cdot y - x \cdot \tau + 18 \cdot \sqrt{y \cdot (81 \cdot y - x \cdot \tau)}) \cdot (x \cdot \tau)^2}}{6 \cdot x \cdot \tau} + \\ + \frac{x \cdot \tau}{3(\sqrt[3]{(162 \cdot y - x \cdot \tau + 18 \cdot \sqrt{y \cdot (81 \cdot y - x \cdot \tau)}) \cdot (x \cdot \tau)^2}} \\ - \frac{1}{6} \approx \sqrt[3]{\frac{3 \cdot y}{2 \cdot x \cdot \tau}}.$$

In the second step, we use an iterative process that refines $|\mathbf{p}_r|_n$ and τ_n in each step in order to determine k. More specifically, we calculate the number $|\mathbf{p}_r|_n$ of outer partitions and the tightening factor τ_n from the previously calculated k_n, starting with $k_0 = 1$. After substituting x and y (cf. Equation (4.3)) in the above equation for k, we obtain the recurrence:

$$k_{n+1} = \sqrt[3]{\frac{3 \cdot n_s}{2 \cdot (c\_io + 2 \cdot c\_cpu) \cdot \tau_n} \cdot \left(\frac{c\_io}{b} + \frac{4 \cdot n_r \cdot c\_cpu}{|\mathbf{p}_r|_n}\right)} \tag{4.4}$$

We start with $k_0 = 1$ and calculate the number of outer partitions, $|\mathbf{p}_r|_0$, according to Lemma 4, i.e.,

$$|\mathbf{p}_r|_n = min(k_n \lceil \lambda_r \cdot k_n \rceil + k_n - \frac{\lceil \lambda_r \cdot k_n \rceil^2}{2} - \frac{\lceil \lambda_r \cdot k_n \rceil}{2}, n_r),$$

and the tightening factor τ_0 as the number of inner partitions (cf. Lemma 4) divided by the number of possible partitions (cf. Proposition 5), i.e.,

$$\tau_n = \frac{min(k_n \lceil \lambda_s \cdot k_n \rceil + k_n - \frac{\lceil \lambda_s \cdot k_n \rceil^2}{2} - \frac{\lceil \lambda_s \cdot k_n \rceil}{2}, n_s)}{(k_n^2 + k_n)/2}.$$

We repeatedly calculate k_{n+1} from $|\mathbf{p}_r|_n$ and τ_n until k converges to the minimum cost.

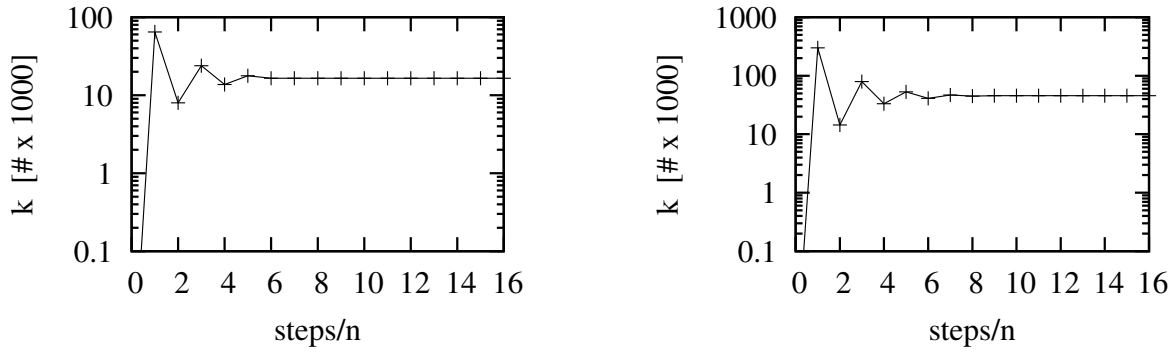
Example 22. Consider two relations $\mathbf{r}$ and $\mathbf{s}$, each with a time range $|U| = 10$M. Relation $\mathbf{r}$ has $n_r = 10$M tuples, and the maximum duration of tuples is $l_r = 1,000$, i.e., $\lambda_r = 0.0001$. Relation $\mathbf{s}$ has $n_s = 100$M tuples, and the maximum duration of tuples is $l_s = 5,000$, i.e., $\lambda_s = 0.0005$. Both relations are stored in main memory in blocks of 512 bytes. With a tuple size of 35 bytes, $b = 14$ tuples fit into a block. The time of a CPU operation is 0.5 nsec (2GHz), and the time

to fetch a block from main memory is 10 nsec, i.e., $c\_cpu = 0.5$ and $c\_io = 10$. Starting with $n = 0$ and $k_0 = 1$, we get the following values:

n	k_n	$\lvert\mathbf{p}_r\rvert_n$	τ_n
0	1	1	1
1	64,633	517,036	0.00105
2	7,967	15,933	0.00126
3	23,819	95,270	0.00109
4	13,761	41,280	0.00116
5	17,795	53,382	0.00112
6	16,522	49,563	0.00121
7	**16,521**	**49,560**	**0.00121**
8	**16,521**	**49,560**	**0.00121**

Thus, k converges to $k = 16,521$, which is the number of granules for the OIPJOIN.

Figure 4.5 illustrates the convergence of k for relations of different size. The iterative process to find k converges since at each step we reduce the power by $\frac{1}{3}$. Note that due to the ceiling function and integer calculus in $\lvert\mathbf{p}_r\rvert_n$ and τ_n, k may not converge to a single number, but oscillate between two. In this case the final k is the average between these two numbers.



(a) $n_r = 10$M and $n_s = 100$M (b) $n_r = 100$M and $n_s = 1$G

Figure 4.5: Convergence of k.

4.6.3 Complexity Analysis

The complexity of the OIPJOIN is composed of three parts: $O(|\mathbf{p}_r| \cdot \mathsf{APA})$ partition fetches; $O(n_s \cdot n_r \cdot \mathsf{AFR})$ false hits; and $O(n_z)$ for retrieving n_z result tuples. After substituting AFR and APA according to Theorem 2 and 3, we get the sum $O(|\mathbf{p}_r| \cdot \tau \cdot k^2) + O(n_s \cdot n_r \cdot \frac{1}{k}) + O(n_z)$. The asymptotic k according to Equation (4.4) is $k = O((\frac{n_s \cdot n_r}{|\mathbf{p}_r| \cdot \tau})^{1/3})$.

The *upper bound complexity* occurs with tightening factor $\tau = 1$ (no tightening). In this case we get a low k to keep the cost for partition accesses low. From $\tau = 1$ we get $|\mathbf{p}_r| = O(k^2)$ (cf. Section 4.5.2), i.e.,

$$\begin{aligned} k &= O((\frac{n_s \cdot n_r}{k^2})^{1/3}) \\ k^{5/3} &= O((n_s \cdot n_r)^{1/3}) \\ k &= O((n_s \cdot n_r)^{1/5}) \end{aligned}$$

Inserting this into the above sum gives $O(n_r^{4/5} \cdot n_s^{4/5} + n_z)$.

The *lower bound complexity* occurs with tightening factor $\tau = O(\frac{1}{k})$ (maximal tightening). In this case we get a high k, since the cost for partition accesses is low. From $\tau = O(\frac{1}{k})$ we get $|\mathbf{p}_r| = O(k)$. Then k is:

$$k = O((\frac{n_s \cdot n_r}{k \cdot \frac{1}{k}})^{1/3}) = O((n_s \cdot n_r)^{1/3})$$

Inserting this into the sum gives $O(n_r^{2/3} \cdot n_s^{2/3} + n_z)$.

To illustrate the complexity, we performed an overlap join between two relations with 5M tuples each and between two relations with 10M tuples each. As a reference point, we also compare it with the lower and upper bound of a sort-merge join (SMJ) of the same relations. Table 4.1 shows the runtimes in seconds (as usual, the time to write result tuples is excluded). We can see that the runtime increased approximately by a factor of $2^{2/3} \cdot 2^{2/3} = 2.52$ for the lower bound and by a factor of $2^{4/5} \cdot 2^{4/5} = 3.03$ for the upper bound. The increase in runtime for the sort-merge join is 2.06 (linear) for the lower bound and 4.00 (quadratic) for the upper bound.

		5M	10M	increase
OIPJOIN:	LB ($\tau \approx 1/k$)	46	120	$\times 2.61$
	UB ($\tau = 1$)	$2,028$	$6,655$	$\times 3.28$
SMJ:	LB	3.2	6.6	$\times 2.06$
	UB	$81,043$	$324,175$	$\times 4.00$

Table 4.1: Runtime and Factor of Runtime Increase.

4.7 Empirical Evaluation

This section evaluates the performance of the OIPJOIN and compares it empirically with the other self-adjusting approaches. The first set of experiments evaluates how k adapts to the cost of CPU operations and the cost of block IOs. We also verify our cost function by relating it to the actual runtime. The second set of experiments evaluates the performance of different approaches for a varying percentage and distribution of long-lived tuples. The ability to efficiently process data with long-lived tuples, i.e., tuples with a non-negligible temporal duration, is the most crucial aspect of algorithms and access methods for temporal data. The OIPJOIN outperforms related approaches by a large margin. The third set of experiments shows that the OIPJOIN scales better than the other approaches for real world datasets, coming from animal feed industry, personnel office, and open source software. Between $0.03\% - 20\%$ of the tuples are larger than 8% of the data's time range, which already leads to significant differences. Finally, we show that the OIPJOIN scales better than the other approaches for disk resident data since it considers both CPU and IO costs.

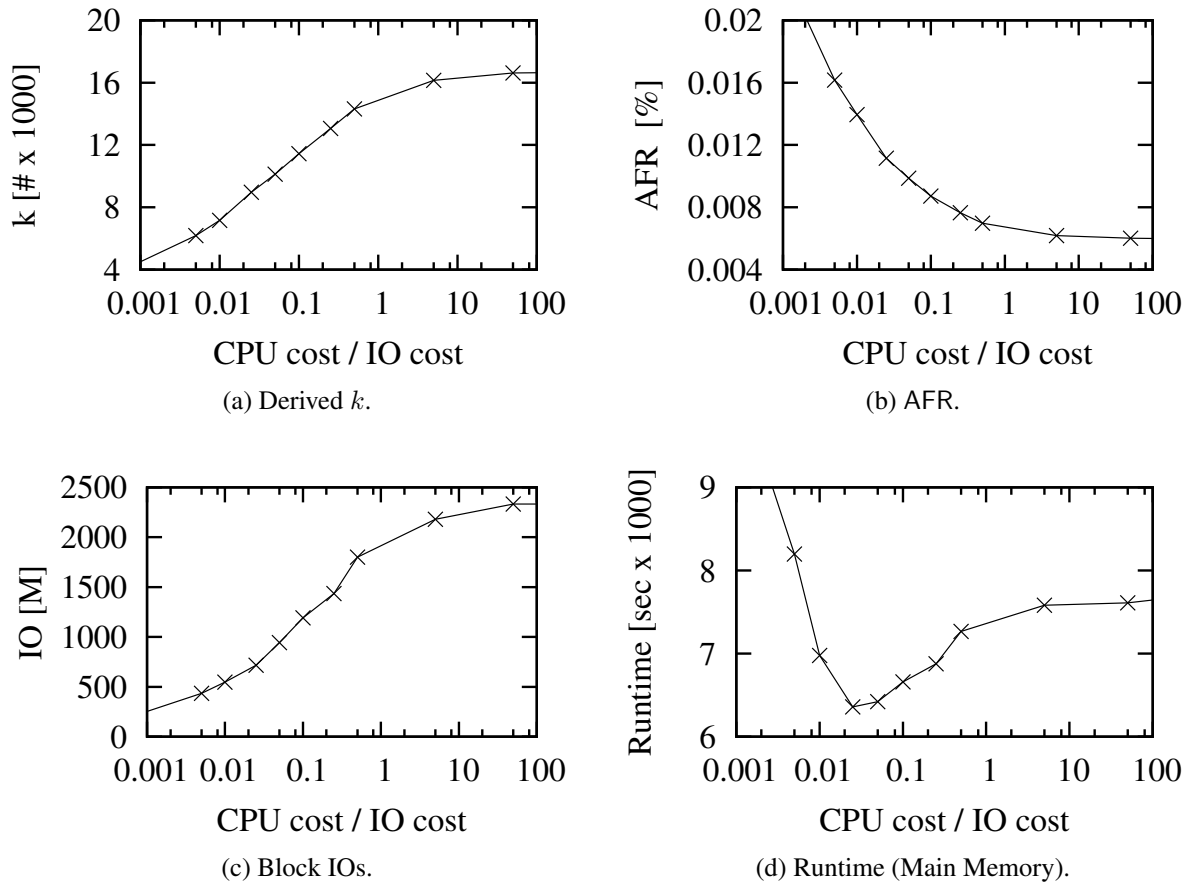
4.7.1 Setup

For the experiments we use a 2 x Intel(R) Xeon(R) CPU E5-2440 0 @ 2.40GHz machine with 64GB main memory running CentOS 6.4. All algorithms have been implemented in C. We use a tuple size of 35 bytes. The block size is 512 bytes for relations stored in main memory (gives the best performance on our machine) and 4K bytes (physical disk block size) when stored on disk. We implemented all algorithms for both disk and main memory storage. The cost to perform a CPU operation (0.5 nsec) on our machine is about 20 times faster than fetching a main memory block (10 nsec). We use synthetic datasets with a time range of $[1, 2^{24}]$ as well as real world datasets (described below).

The OIPJOIN is compared against the following state-of-the-art approaches. *Loose quadtree (lqt)*: We implemented a partition-based algorithm that joins every node of the outer tree with all relevant nodes in the inner tree. We use a cell expansion factor $p = 1$, which is widely accepted as the best value [Sam05; Ulr00] and gave the best results in our experiments. We use density based splitting, i.e., tuples are propagated down the tree only if a block is full. Together with block storage, this gave a runtime improvement up to 400% compared to random access to single tuples. Since in all experiments the loose quadtree outperformed the quadtree, the latter is not shown in the plots. *Relational interval tree (rit)*: We implemented the RI-Tree Up-Down partition-based algorithm [EHS04]. When data is stored in main memory, we do not use blocks to store tuples contiguously. The reason is that even for a clustering index, the time to fetch 512 bytes that contain only a few matching tuples outweighs the advantages of contiguous memory access. *Segment tree (sgt)*: We implemented the segment tree, where the index is build on the inner relation and the overlap join is computed by joining each tuple of the outer relation with the segment tree. Duplicates are identified during join processing by testing whether the intersecting interval starts before the currently joined segment; if so, it has already been joined in a previous segment. *Sort-merge join (smj)*: We implemented a sort-merge join that sorts the tuples of the outer relation by the end point and the tuples of the inner relation by the start point. The sort order of the inner relation is used to stop scanning when an inner tuple has a larger start point than the end point of the outer tuple. The sort order of the outer relation allows to limit the backtracking to the maximum duration of tuples in the relations. We implemented the join using blocks. In spite of more false hits, this increases the performance due to less backtracking. All runtime experiments include the time to create the indices. For all approaches, the index creation time is $\approx 1\%$ of the total runtime for data kept in memory and $\approx 5\%$ for disk resident data.

4.7.2 Number of Granules k

The first experiment shows how the OIPJOIN adapts to c_cpu and c_io. We use synthetic data: an outer relation with 10M tuples and an inner relation with 100M tuples, both with tuple durations up to 0.1% of the time range. Figure 4.6a shows k when varying the ratio $\frac{c\_cpu}{c\_io}$ from 0.001 to 100. When c_cpu gets more expensive, k increases so that more partitions are generated. Figure 4.6b and 4.6c show, respectively, the corresponding AFR (decreasing) and the number of block IOs (increasing). Figure 4.6d shows the runtime for main memory resident data. It illustrates that also for data that is stored in main memory the performance can be increased if the costs of memory IOs and the costs of CPU operations are considered for determining the optimal k.



(a) Derived k.

(b) AFR.

(c) Block IOs.

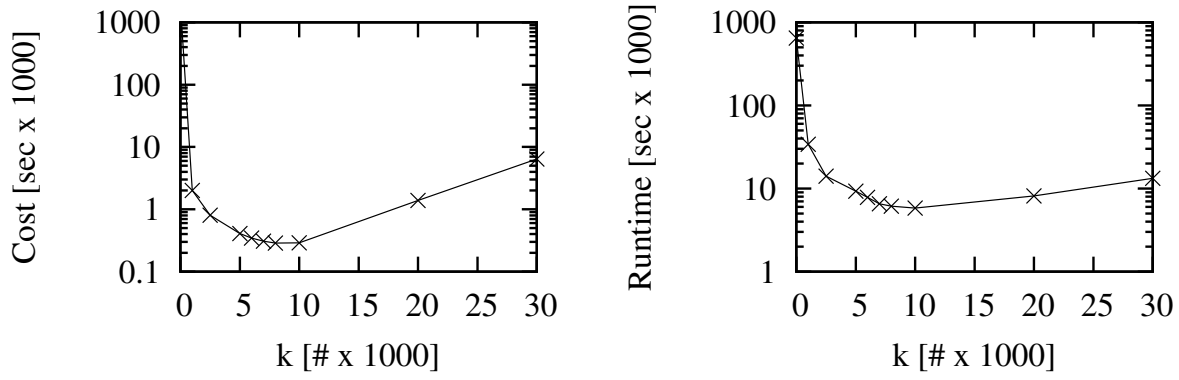
(d) Runtime (Main Memory).

Figure 4.6: Derived k with Varying c_cpu and c_io.

The next experiment compares the cost function of the OIPJOIN to the actual runtime. We use the same relations as in the previous experiment and vary k. Figure 4.7a shows the cost function for $c\_cpu = 0.5$ nsec and $c\_io = 10$ nsec. Figure 4.7b shows the actual runtime for the same setting. It is easy to see that both functions have the same shape with the minimum at $k = 10,130$.

4.7.3 Long-Lived Tuples

The next experiment compares the performance of the OIPJOIN (oip) with the loose quadtree (lqt), the relational interval tree (rit), the segment tree (sgt), and the sort-merge join (smj), by varying the number of long-lived tuples and the maximum duration of tuples. The two input relations have 10M tuples each, with long-lived tuples that have a duration up to 8% and an average duration of 4% of the relation's time range. Short-lived tuples have a maximum duration



(a) Cost Function (Overhead). (b) Runtime.

Figure 4.7: Cost Function and Runtime.

of 0.01% of the time range. Figure 4.8 and Figure 4.9 show the runtime and the AFR of the four algorithms. The AFR of the relational interval tree and segment tree are omitted since they produce no false hits. The OIPJOIN significantly outperforms the other approaches since it does not suffer from long-lived tuples and has a very small AFR (the curve is close to the x-axis). In contrast, the loose quadtree is very sensitive to long-lived tuples, and the AFR increases drastically. This yields much higher runtimes due to excessive comparison operations and the filtering of false hits. Although the relational interval tree does not produce false hits, its performance decreases with the increase of long-lived tuples since a higher number of index nodes need to be joined, which requires a high number of operations on the indices. The segment tree scales worse than the relational interval tree, since with longer tuple durations many duplicates need to be fetched and tested. In our experiments, the segment tree outperforms the relational interval tree only for tuple durations smaller than 0.001%. The performance of the sort-merge join is highly affected by the longest tuple in the dataset, but it scales better than the loose quadtree.

4.7.4 Real World Datasets

We use three real world datasets that differ in size and data distribution. The main properties of these datasets are summarized in Table 4.2. The Incumbent dataset [GSSY98] records the history of employees assigned to projects over a 16 year period at a granularity of days. The Feed dataset records the history of measured nutritive values of feeds over a 24 year period at a granularity of days; a measurement of a nutrient remains valid until a new measurement for the same nutritive value and feed becomes available. The Webkit dataset [web12] records the history of files of

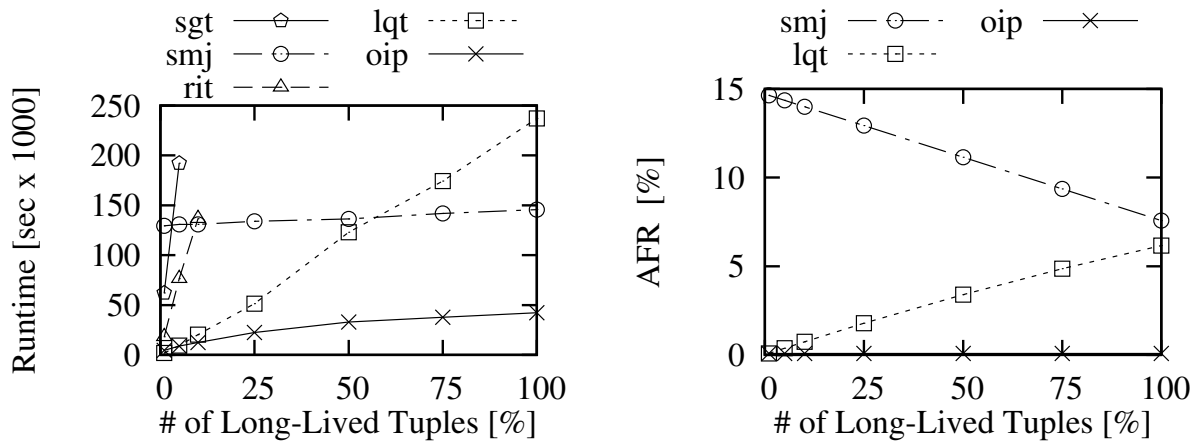


Figure 4.8: Varying Number of Long-Lived Tuples.

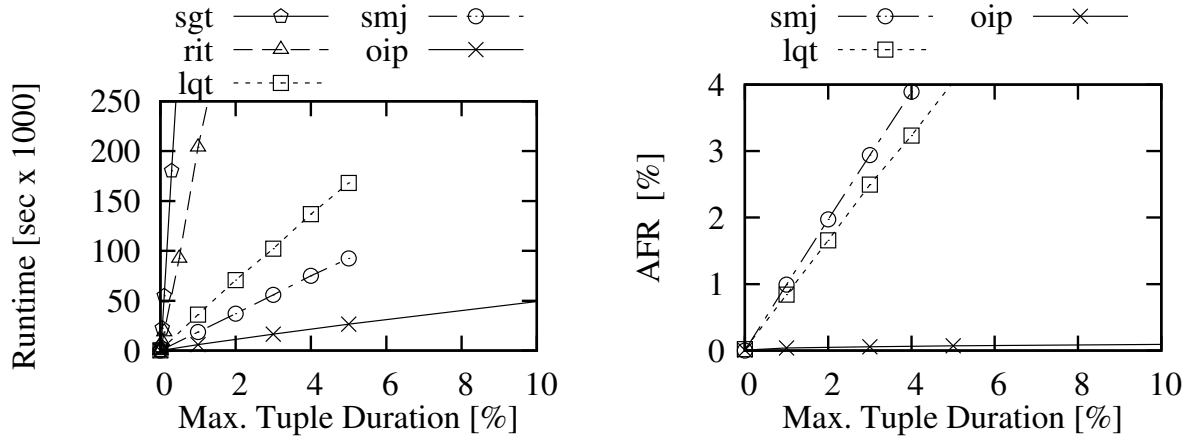


Figure 4.9: Varying Maximum Duration of Tuples.

the svn repository of the Webkit project over a 11 year period at a granularity of milliseconds. The valid times indicate the periods when a file did not change. Figure 4.10 shows the temporal distributions of the data (i.e., the number of overlapping tuple intervals at each time point) and the histograms of tuple durations.

For all three datasets we perform an overlap join, using a subset of the dataset as the outer relation and the entire dataset as the inner relation. We use the smaller as the outer relation, since it typically has fewer partitions, and thus some partitions of the larger relation are not accessed at all during the join. Figure 4.11 shows the runtime and the AFR for the three datasets depending on the size of the outer relation. The OIPJOIN has the best performance in all three settings. The other approaches suffer from long-lived tuples, e.g., the AFR of the loose quadtree is much

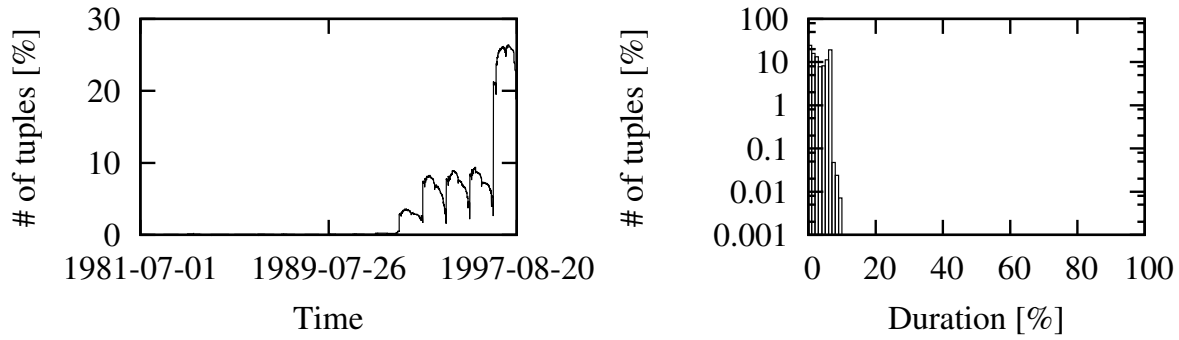
	Incumbent	**Feed**	**Webkit**
Cardinality	$83,852$	$3,697,957$	$1,213,476$
Time Range	$5,895$	$8,610$	$\approx 2^{39}$
Min. Duration	1	1	$\approx 2^{10}$
Max. Duration	574	$8,589$	$\approx 2^{39}$
Avg. Duration	184	432	$\approx 2^{34}$
Distinct Points	$2,689$	$5,584$	$110,165$

Table 4.2: Properties of Real World Datasets.

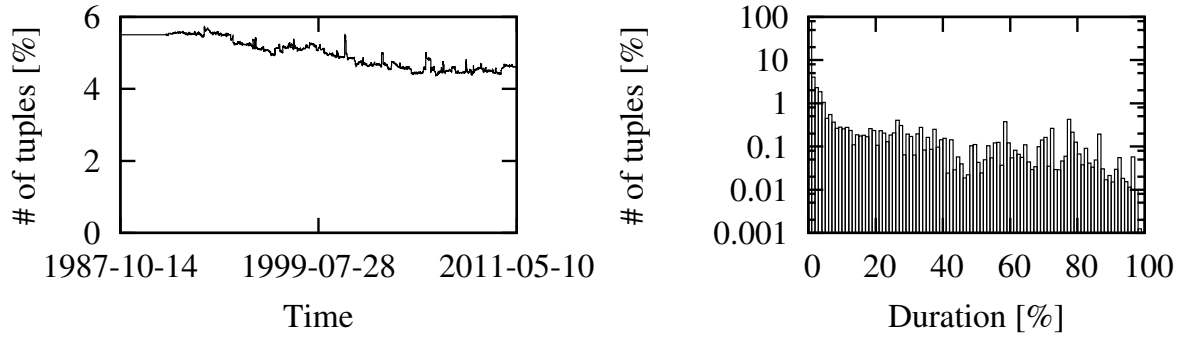
larger than the one of the OIPJOIN, and it does not adapt to the size of the dataset. The AFR of the sort-merge join is omitted since it reaches 30–50%.

4.7.5 Scalability on Disk

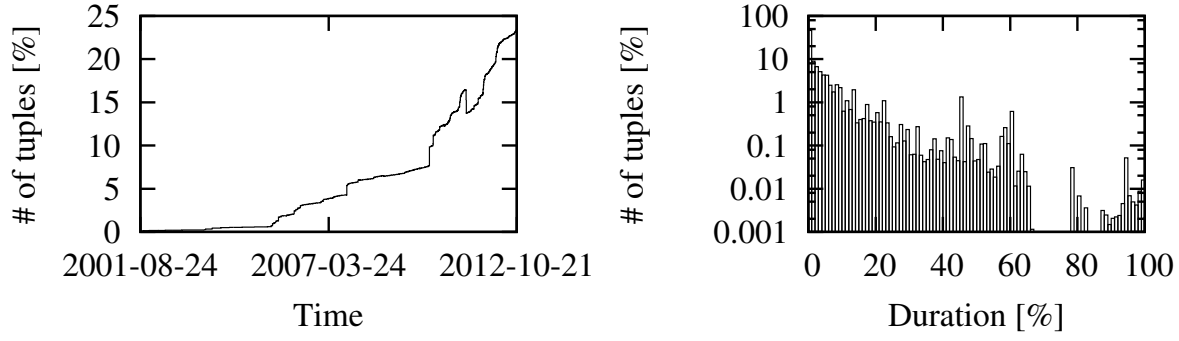
The last experiment shows the scalability of the algorithms for disk resident data. We vary the number of inner tuples from 100M to 1500M. The number of outer tuples is 1% of the inner relation. Both relations have tuple durations up to 0.1% of the time range. c_io is 200 times higher than c_cpu. Figures 4.12a and 4.12b show the number of block IOs and the AFR. Figure 4.12c shows the runtime behavior on a server with 64GB of main memory, where a large number of disk blocks is cached by the operating system. Although the loose quadtree, due to its density-based splitting strategy, is the best approach in terms of block IOs, it produces a large number of false hits. The OIPJOIN adapts to both the cost of block IOs and the cost of false hits, and thus outperforms all other approaches in terms of runtime. The segment tree performs worst, in particular in terms of IO (close to the y-axis), since for each outer tuple, duplicated inner tuples and thus disk blocks are fetched several times. We run the same experiment for the three best approaches on a different machine with a similar CPU but only 4GB main memory, that is, fewer disk blocks are cached by the operating system. The runtime behavior is shown in Figure 4.12d and is slower, as expected. The loose quadtree performs much worse on this machine despite fewer IOs. The reason is that the OIPJOIN and the sort-merge join benefit from sequential reads due to sorting. The loose quadtree does not have sequential blocks on disk, hence the disk seek time deteriorates the performance of the loose quadtree.



(a) Incumbent Dataset.

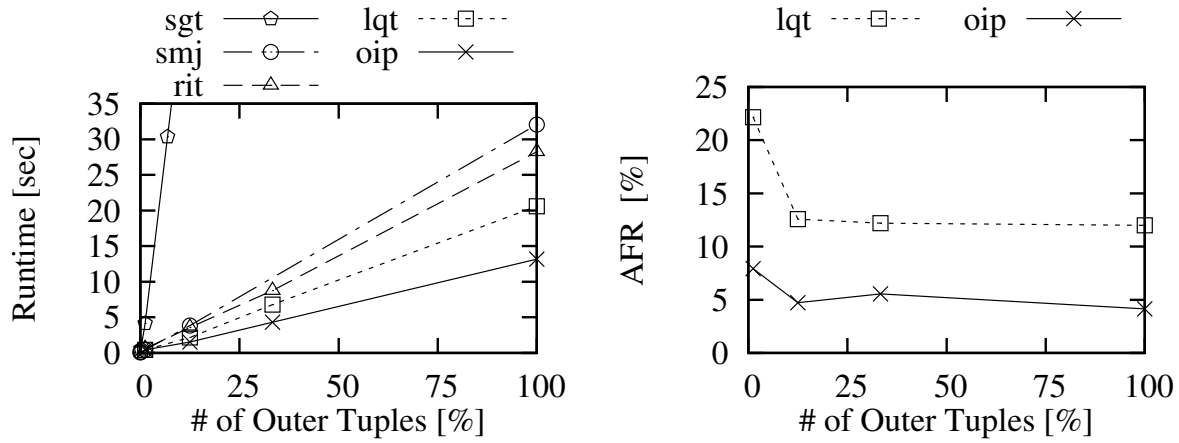


(b) Feed Dataset.

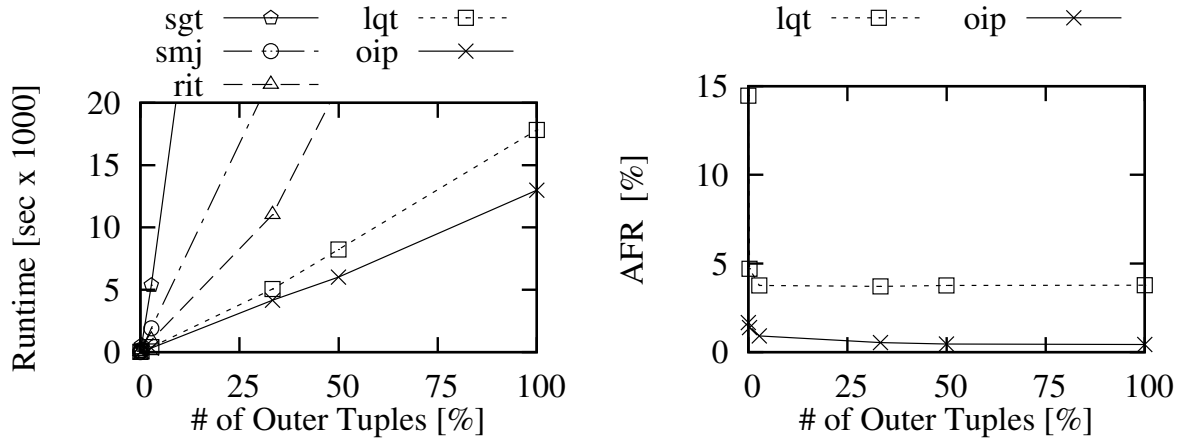


(c) Webkit Dataset.

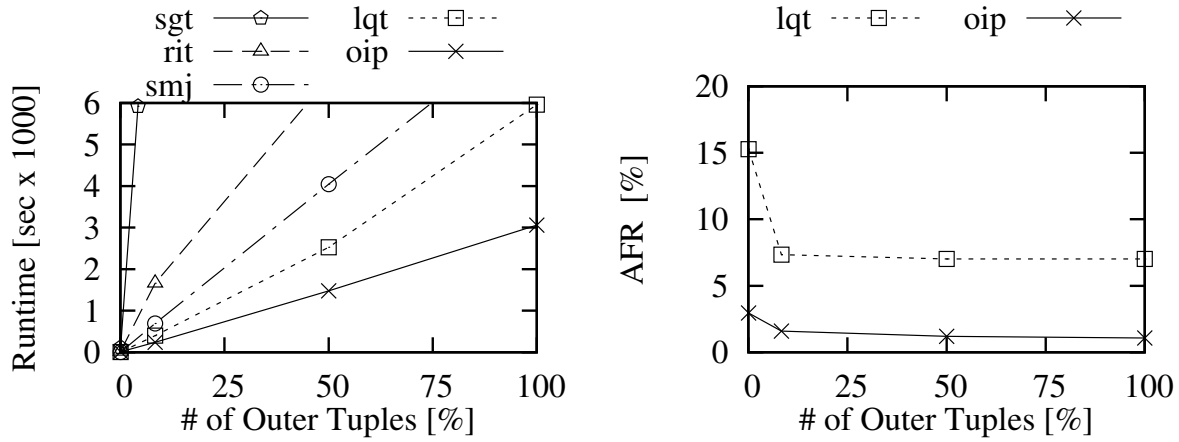
Figure 4.10: Tuple Intervals per Time Point and Duration Histogram of Real World Datasets.



(a) Incumbent Dataset.

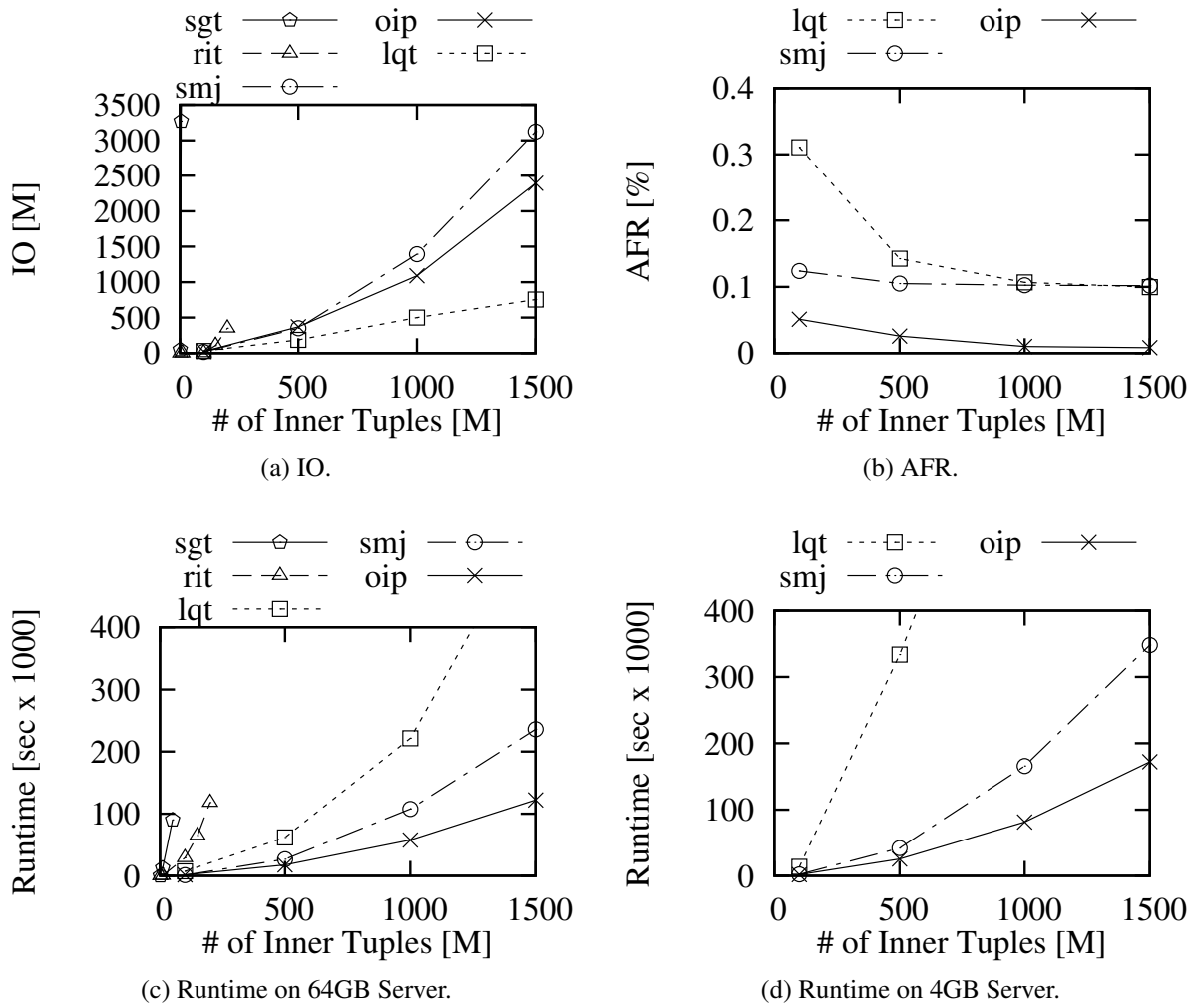


(b) Feed Dataset.



(c) Webkit Dataset.

Figure 4.11: Runtime and AFR for Real World Datasets.



(a) IO.

(b) AFR.

(c) Runtime on 64GB Server.

(d) Runtime on 4GB Server.

Figure 4.12: Varying Number of Tuples of Disk Resident Data.

4.7.6 Summary

The OIPJOIN is the most efficient and robust approach if the data includes long-lived tuples since it provides a constant clustering guarantee and adapts to both the cost for false hits and the cost for partition accesses. The loose quadtree and the relational interval tree are only competitive if the dataset contains a very low number of long-lived tuples. In all other cases, either the false hits or the navigation in the index structure incur high costs. For datasets with only very short tuples (or point data), the sort-merge join is the most efficient approach, but it deteriorates as soon as the dataset contains a few long-lived tuples.

4.8 Conclusion

In this paper, we have presented the overlap interval partition join (OIPJOIN) for valid-time relation together with *Overlap Interval Partitioning* ($\mathcal{OIP}$). $\mathcal{OIP}$ permits overlapping partitions that are not derived from a recursive hierarchical space division. In contrast to other approaches, the OIPJOIN does not deteriorate in the presence of long-lived tuples and adjusts the number of partitions based on the size of the dataset, the cost of CPU operations, and the cost of IOs. An in-depth empirical evaluation shows that the OIPJOIN outperforms state-of-the-art techniques for the overlap join.

Future work points in several directions. First, it is interesting to investigate how to update $\mathcal{OIP}$ incrementally if the relation changes, since the partitioning allows an expansion on both space boundaries by increasing k and maintaining an offset on the indices. For this the effects on k and the treatment of the ending variable "now" must be studied. Second, it is possible to refine the cost function for, e.g., different buffer replacement strategies. Third, we have planned to develop statistics to tighten k not only based on the maximum duration of tuples, but also on the data distribution.

CHAPTER 5

Conclusion and Future Work

In this thesis, we propose an approach that fully supports the querying of data with time intervals in relational database systems. Our approach is based on two temporal primitives, a temporal aligner and a temporal normalizer, that align the intervals of tuples in a relation. Reduction rules map, with the help of a temporal primitive, a temporal operator to a traditional non-temporal operator that uses equality on aligned intervals. We prove that the result of the temporal operators is snapshot reducible and respects lineage information.

Our approach allows operators to access the original time intervals in predicates and functions, such as join conditions and aggregation functions, by using timestamp propagation, and to scale attribute values that are interval-dependent with the help of user-defined functions. We implemented the temporal primitives into the kernel of the open source database system PostgreSQL that allows to leverage existing algorithms and query optimization techniques. We show how the scaling of attribute values that are interval-dependent, such as project budgets or costs, can be achieved at query time using simple user-defined functions.

The implementation of the temporal primitives internally relies on efficiently finding pairs of matching tuples with overlapping intervals. For cases when current join algorithms do not perform well, we propose the overlap interval partition join (OIPJOIN). The OIPJOIN is based on

overlap interval partitioning ($\mathcal{OIP}$), a partitioning approach that, due to its constant clustering guarantee, does not deteriorate in performance when the data has long intervals. To make the OIPJOIN self-adjusting, i.e., parameter-free, we derive the optimal parameter k of $\mathcal{OIP}$ for the OIPJOIN, by minimizing its cost function for false hits and partition accesses based on CPU and IO costs. Extensive experiments show that our approach outperforms state-of-the-art approaches.

Future Work We currently limit the temporal operators and temporal primitives to duplicate free temporal relations, i.e., relations where no value-equivalent tuple with overlapping time interval exists. This has been widely adopted in temporal database research. It is interesting future work to further extend the operators and temporal primitives for relations that are not duplicate free.

Currently, the temporal primitives are general for group and tuple based operators. In order to improve efficiency we plan to customize the temporal primitives for specific temporal operators to not produce aligned tuples that do not contribute to the final result.

We will also investigate more advanced statistics for $\mathcal{OIP}$ to calculate the reduced average number of partition accesses APA, for cases when the datasets are skewed or do not contain tuples for all durations. This can for instance be achieved with the help of histograms on duration and position of tuples. Furthermore, we will refine the cost function of the OIPJOIN for different buffer replacement strategies when the data is stored on disk.

Bibliography

[ABPT01] Mikkel Agesen, Michael H. Böhlen, Lasse Poulsen, and Kristian Torp. A split operator for now-relative bitemporal databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 41–50, Washington, DC, USA, 2001. IEEE Computer Society.

[AHVdB96] Serge Abiteboul, Laurent Herr, and Jan Van den Bussche. Temporal versus first-order logic to query temporal databases. In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '96, pages 49–57, New York, NY, USA, 1996. ACM.

[BBJS97] John Bair, Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Notions of upward compatibility of temporal query languages. *Wirtschaftsinformatik*, 39(1):25–34, 1997.

[BCN89] Andries E. Brouwer, Arjeh M. Cohen, and Arnold Neumaier. *Distance Regular Graphs*. Springer-Verlag, 1989.

[Ben77] Jon Louis Bentley. Solutions to klee's rectangle problems. Technical report, Carnegie Mellon University, 1977.

[BGJ06a] Michael Böhlen, Johann Gamper, and Christian S. Jensen. Multi-dimensional aggregation for temporal data. In *Proceedings of the 10th international conference on*

Advances in Database Technology, EDBT'06, pages 257–275, Berlin, Heidelberg, 2006. Springer-Verlag.

[BGJ06b] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. An algebraic framework for temporal attribute characteristics. *Ann. Math. Artif. Intell.*, 46(3):349–374, 2006.

[BGJS09] Michael H. Böhlen, Johann Gamper, Christian S. Jensen, and Richard T. Snodgrass. Sql-based temporal query languages. In Liu and Özsu [LÖ09], pages 2762–2768.

[BJ02] Michael H. Böhlen and Christian S. Jensen. *Encyclopedia of Information Systems*, chapter Temporal Data Model and Query Language Concepts. Academic Press, 2002.

[BJ09] Michael H. Böhlen and Christian S. Jensen. Sequenced semantics. In Liu and Özsu [LÖ09], pages 2619–2621.

[BJS95] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Evaluating the completeness of tsql2. In *Proceedings of the International Workshop on Temporal Databases: Recent Advances in Temporal Databases*, pages 153–172, London, UK, UK, 1995. Springer-Verlag.

[BJS00] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Temporal statement modifiers. *ACM Trans. Database Syst.*, 25(4):407–456, December 2000.

[BJS09a] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Nonsequenced semantics. In Liu and Özsu [LÖ09], pages 1913–1915.

[BJS09b] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Temporal compatibility. In Liu and Özsu [LÖ09], pages 2936–2939.

[BKOS00] Mark Berg, Marc Kreveld, Mark Overmars, and OtfriedCheong Schwarzkopf. More geometric data structures. In *Computational Geometry*, pages 211–233. Springer Berlin Heidelberg, 2000.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM.

[BS09] Norbert Beckmann and Bernhard Seeger. A revised r*-tree in comparison with related index structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 799–812, New York, NY, USA, 2009. ACM.

[BT09] Sarah Cohen Boulakia and Wang Chiew Tan. Provenance in scientific databases. In Liu and Özsu [LÖ09], pages 2202–2207.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, June 2000.

[Dav11] Jeff Davis. Online temporal PostgreSQL reference. http://temporal.projects.postgresql.org/reference.html, 2011.

[DBG12] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal alignment. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 433–444, New York, NY, USA, 2012. ACM.

[DBG13] Anton Dignös, Michael Böhlen, and Johann Gamper. Query time scaling of attribute values in interval timestamped databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 1304–1307, Washington, DC, USA, 2013. IEEE Computer Society.

[DD02] Chris Date and Hugn Darwen. *Temporal Data and the Relational Model*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[EHS04] Jost Enderle, Matthias Hampel, and Thomas Seidl. Joining interval data in relational databases. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 683–694, New York, NY, USA, 2004. ACM.

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[GBJ09] Johann Gamper, Michael H. Böhlen, and Christian S. Jensen. Temporal aggregation. In Liu and Özsu [LÖ09], pages 2924–2929.

[GJSS05] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. Join operations in temporal databases. *VLDB J.*, 14(1):2–29, 2005.

[Gro12] PostgreSQL Global Development Group. Documentation manual PostgreSQL – range types. http://www.postgresql.org/docs/9.2/static/rangetypes.html, 2012.

[GSSY98] Jose A. G. Gendrano, R. Shah, Richard T. Snodgrass, and Jun Yang. University information system (uis) dataset. TimeCenter CD-1, 1998.

[JS09] Christian S. Jensen and Richard T. Snodgrass. Timeslice operator. In Liu and Özsu [LÖ09], pages 3120–3121.

[JSS94] Christian S. Jensen, Michael D. Soo, and Richard T. Snodgrass. Unifying temporal data models via a conceptual model. *Inf. Syst.*, 19(7):513–547, October 1994.

[KM12] Krishna G. Kulkarni and Jan-Eike Michels. Temporal features in sql: 2011. *SIGMOD Record*, 41(3):34–43, 2012.

[KPS00] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *Proceedings of 26th International Conference on Very Large Data Bases*, VLDB 2000, pages 407–418, 2000.

[KS95] Nick Kline and Richard T. Snodgrass. Computing temporal aggregates. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 222–231, Washington, DC, USA, 1995. IEEE Computer Society.

[KS97] Nick Koudas and Kenneth C. Sevcik. Size separation spatial join. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 324–335, New York, NY, USA, 1997. ACM.

[LM97] Nikos A. Lorentzos and Yannis G. Mitsopoulos. Sql extension for interval data. *IEEE Trans. on Knowl. and Data Eng.*, 9(3):480–499, May 1997.

[LÖ09] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009.

[Lor09] Nikos A. Lorentzos. Period-stamped temporal models. In Liu and Özsu [LÖ09], pages 2094–2098.

[LOT94] Hongjun Lu, Beng Chin Ooi, and Kian-Lee Tan. On spatially partitioned temporal join. In *Proceedings of the 20th International Conference on Very Large Data*

Bases, VLDB '94, pages 546–557, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[LR96] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 247–258, New York, NY, USA, 1996. ACM.

[LSD+01] Wei Li, Richard T. Snodgrass, Shiyan Deng, Vineel Kumar Gattu, and Aravindan Kasthurirangan. Efficient sequenced integrity constraint checking. In Dimitrios Georgakopoulos and Alexander Buchmann, editors, *ICDE*, pages 131–140. IEEE Computer Society, 2001.

[MFVLI03] Bongki Moon, Ines Fernando Vega Lopez, and Vijaykumar Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):744–759, March 2003.

[Mur08] Chuck Murray. *Oracle Database Workspace Manager Developer's Guide*. Oracle Corporation, 2008. http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28396.pdf.

[NTH+13] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. Touch: In-memory spatial join by hierarchical data-oriented partitioning. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 701–712, New York, NY, USA, 2013. ACM.

[Sam05] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[SE96] Daeweon Son and Ramez Elmasri. Efficient temporal join processing using time index. In *Proceedings of the Eighth International Conference on Scientific and Statistical Database Management*, SSDBM '96, pages 252–261, Washington, DC, USA, 1996. IEEE Computer Society.

[Seg93] Arie Segev. Join Processing and Optimization in Temporal Relational Databases. In Abdullah Uz Tansel, James Clifford, Shashi K. Gadia, Sushil Jajodia, Arie Segev, and Richard T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 15, pages 356–387. Benjamin/Cummings Publishing Company, 1993.

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[SJS95] Michael D. Soo, Christian S. Jensen, and Richard T. Snodgrass. An algebra for tsql2. In *The TSQL2 Temporal Query Language*, pages 501–544. Kluwer, 1995.

[SNG12] Cynthia M. Saracco, Matthias Nicola, and Lenisha Gandhi. *A matter of time: Temporal data management in DB2 10*. IBM Corporation, 2012. http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/dm-1204db2temporaldata-pdf.pdf.

[Sno95] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, 1995.

[Sno00] Richard T. Snodgrass. *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[SSA13] Hanan Samet, Jagan Sankaranarayanan, and Michael Auerbach. Indexing methods for moving object databases: Games and other applications. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 169–180, New York, NY, USA, 2013. ACM.

[SSJ94] Michael D. Soo, Richard T. Snodgrass, and Christian S. Jensen. Efficient evaluation of the valid-time natural join. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 282–292, Washington, DC, USA, 1994. IEEE Computer Society.

[Ter10] Teradata. *Teradata Database Temporal Table Support*. Teradata Corporation, 2010. http://www.info.teradata.com/edownload.cfm?itemid=102320064.

[TK95] Vassilis J. Tsotras and Nickolas Kangerlaris. The snapshot index: An i/o-optimal access method for timeslice queries. *Inf. Syst.*, 20(3):237–260, 1995.

[Tom96] David Toman. Point vs. interval-based query languages for temporal databases. In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '96, pages 58–67, New York, NY, USA, 1996. ACM.

[Tom97] David Toman. Point-based temporal extensions of sql and their efficient implementation. In *Temporal Databases, Dagstuhl*, pages 211–237, 1997.

[TS04] Paolo Terenziani and Richard T. Snodgrass. Reconciling point-based and interval-based semantics in temporal relational databases: A treatment of the telic/atelic distinction. *IEEE Trans. Knowl. Data Eng.*, 16(5):540–551, 2004.

[Ulr00] Thatcher Ulrich. Loose octrees. In *Game Programming Gems*, pages 444–453. Charles River Media, 2000.

[VLSM05] Ines Fernando Vega Lopez, Richard T. Snodgrass, and Bongki Moon. Spatiotemporal aggregate computation: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 17(2):271–286, February 2005.

[web12] The webkit open source project. http://www.webkit.org, 2012.

[YW03] Jun Yang and Jennifer Widom. Incremental computation and maintenance of temporal aggregates. *The VLDB Journal*, 12(3):262–283, October 2003.

[ZMT+01] Donhui Zhang, Alexander Markowetz, Vassilis Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. Efficient computation of temporal aggregates with range predicates. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 237–245, New York, NY, USA, 2001. ACM.

[ZTS02] Donghui Zhang, Vassilis J. Tsotras, and Bernhard Seeger. Efficient temporal join processing using indices. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 103–113, Washington, DC, USA, 2002. IEEE Computer Society.